HETEROGENEOUS PARALLEL VIRTUAL MACHINE: A PORTABLE PROGRAM REPRESENTATION AND COMPILER FOR PERFORMANCE AND ENERGY OPTIMIZATIONS ON HETEROGENEOUS PARALLEL SYSTEMS

BY

MARIA KOTSIFAKOU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Doctoral Committee:

Professor Vikram S. Adve, Chair
Professor Sarita V. Adve
Assistant Professor Sasa Misailovic
Professor David M. Brooks, Harvard University

ABSTRACT

Programming heterogeneous parallel systems, such as the SoCs (System-on-Chip) on mobile and edge devices is extremely difficult; the diverse parallel hardware they contain exposes vastly different hardware instruction sets, parallelism models and memory systems. Moreover, a wide range of diverse hardware and software approximation techniques are available for applications targeting heterogeneous SoCs, further exacerbating the programmability challenges. In this thesis, we alleviate the programmability challenges of such systems using flexible compiler intermediate representation solutions, in order to benefit from the performance and superior energy efficiency of heterogeneous systems.

First, we develop Heterogeneous Parallel Virtual Machine (HPVM), a parallel program representation for heterogeneous systems, designed to enable functional and performance portability across popular parallel hardware. HPVM is based on a *hierarchical dataflow graph* with side effects. HPVM successfully supports three important capabilities for programming heterogeneous systems: a compiler intermediate representation (IR), a virtual instruction set (ISA), and a basis for runtime scheduling. We use the HPVM representation to implement an HPVM prototype, defining the HPVM IR as an extension of the Low Level Virtual Machine (LLVM) IR. Our results show comparable performance with optimized OpenCL kernels for the target hardware from a single HPVM representation using translators from HPVM virtual ISA to native code, IR optimizations operating directly on the HPVM representation, and the capability for supporting flexible runtime scheduling schemes from a single HPVM representation.

We extend HPVM to ApproxHPVM, introducing *hardware-independent approximation metrics in the IR* to enable maintaining accuracy information at the IR level and mapping of application-level end-to-end quality metrics to system level "knobs". The approximation metrics quantify the acceptable accuracy loss for individual computations. Application programmers only need to specify high-level, and end-to-end, quality metrics, instead of detailed parameters for individual approximation methods. The ApproxHPVM system then automatically tunes the accuracy requirements of individual computations and maps them to approximate hardware when possible. ApproxHPVM results show significant performance and energy improvements for popular deep learning benchmarks.

Finally, we extend to ApproxHPVM to ApproxTuner, a compiler and runtime system for approximation. ApproxTuner extends ApproxHPVM with a wide range of hardware and software approximation techniques. It uses a *three step approximation tuning strategy*, a

combination of development-time, install-time, and dynamic tuning. Our strategy ensures software portability, even though approximations have highly hardware-dependent performance, and enables efficient dynamic approximation tuning despite the expensive offline steps. ApproxTuner results show significant performance and energy improvements across 7 Deep Neural Networks and 3 image processing benchmarks, and ensures that high-level end-to-end quality specifications are satisfied during adaptive approximation tuning.

To my family, for their love and support.
Especially to my little brother, who I love and miss greatly.

ACKNOWLEDGMENTS

There are many people that have supported me and made it possible for me to succeed during my Ph.D., and for that I am most grateful. First and foremost, I would like to thank my advisor, Professor Vikram Adve, for being my mentor during this important time, for investing time and effort in training me, and for asking all the hard questions as the closest collaborator to most research projects I was involved in. I would like to thank Assistant Professor Sasa Misailovic and Professor Sarita Adve for closely collaborating on ApproxHPVM and ApproxTuner. Finally, I want to thank all members of my Ph.D. committee for their valuable advice and insights towards improving this thesis.

I would like to thank Prakalp Srivastava for collaborating with me on Heterogeneous Parallel Virtual Machine (HPVM) and ApproxHPVM, and Hashim Sharif for collaborating with me on ApproxHPVM and ApproxTuner. My collaboration with them has been a very rewarding experience. I am thankful to all my co-authors on these projects: Muhammad Huzaifa, Keyur Joshi, Rakesh Komuravelli, Akash Kothari, Yasmin Sarita, Ben Schreiber, Matt Sinclair, Elizabeth Wang, Nathan Zhao, Yifan Zhao.

I want to thank all my lab mates in the research group of Prof. Vikram Adve, for providing a fun work environment and for the pleasure of discussing our research projects from multiple different perspectives that would help inspire solutions and work through problems. I am especially grateful to Hashim Sharif and Adel Ejjeh.

I want to thank my boyfriend, Theo, who has been with me in the same research group since the start of my Ph.D., and has supported me during many difficult and stressful times.

My work has been supported by the National Science Foundation (NSF) under grant CCF 13-02641, by the Center for Future Architectures Research (C-FAR) - one of six centers of Semiconductor Technology Advanced Research network (STARnet), a Semiconductor Research Corporation (SRC) program sponsored by Microelectronics Advanced Research Corporation (MARCO) and Defense Advanced Research Projects Agency (DARPA). Part of this work was also supported by DARPA through the Domain Specific System on Chip (DSSoC) program.

Most of all, I want to thank my family. Each of them, in their own unique way, was always there for me even though we live far away. I specifically want to mention my little brother, who was always proud of me, and although he is now somewhere I can't reach him I hope to see him again some day. For their unconditional love and support during my Ph.D. and all my life, I am truly grateful and I am proud to dedicate this dissertation to them.

Finally, I want to express my deepest gratitude to Prof. Vikram Adve for his incredible understanding, patience and support through a very difficult time for me, my brother and my family. For helping me through that time and always being there for me when I needed someone to talk to, I am truly thankful to Theo, as well as Adel and Hashim.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
1.1 Heterogeneous Systems 1
1.2 Programmability Challenges of Heterogeneous Systems 3
1.3 Summary of Related Work 8
1.4 Approach 9
1.5 Contributions 17
1.6 Thesis Organization 19

CHAPTER 2 RELATED WORK 20
2.1 Programming Technologies for Heterogeneous Systems 20
2.2 Approximate Computing 25

CHAPTER 3 THE HPVM REPRESENTATION 29
3.1 HPVM Program 29
3.2 Dataflow Node 30
3.3 Dataflow Edge 32
3.4 Input and Output Bind 34
3.5 Integration with Host Code 34
3.6 HPVM Representation of Major Parallel Hardware Features 35

CHAPTER 4 THE HPVM SYSTEM 39
4.1 HPVM Virtual ISA and Compiler IR 39
4.2 Compilation Strategy 45
4.3 HPVM Runtime and Scheduling Framework 51
4.4 Compiler Optimization 53

CHAPTER 5 HPVM EVALUATION 56
5.1 Experimental Setup and Benchmarks 56
5.2 Portability and Comparison with Hand Tuning 57
5.3 Evaluation of Flexible Scheduling 58
5.4 Node Fusion Optimization Evaluation 60
5.5 Conclusion 60

CHAPTER 6 APPROXHPVM 62
6.1 The ApproxHPVM Intermediate Representation 62
6.2 System Workflow 66
6.3 Code Generation 67
6.4 Evaluation 72

CHAPTER 7 APPROXTUNER: A COMPILER AND RUNTIME SYSTEM FOR ADAPTIVE APPROXIMATIONS 76
7.1 Motivating Example 76
7.2 ApproxTuner 78
7.3 Methodology 87
7.4 Evaluation 90
7.5 Sensitivity to Approximation Techniques 96

CHAPTER 8 CONCLUSION AND FUTURE DIRECTIONS 98
8.1 Conclusion 98
8.2 Future Directions 99

REFERENCES 102

CHAPTER 1: INTRODUCTION

1.1 HETEROGENEOUS SYSTEMS

The combination of Moore's law and Denard's scaling enabled computer architects to develop new generations of single core processors that were faster compared to their predecessors due to containing a larger number of smaller transistors. Smaller transistors have higher operating frequency while power density remained constant. With the end of Denard's scaling, as the size of the transistors decreased the power density increased, thus making it impossible to design single core processors with the same power density. Moore's Law continued to apply, but computer architects turned to multi-core architectures in order to utilize the available number of transistors [1]. However, this approach is eventually limited by power constraints [2], which eventually lead to the slowdown of Moore's Law.

The slow down of Moore's law and the end of Denard's scaling has given rise to custom hardware components that deviate from the single core processors contained in traditional homogeneous systems. Heterogeneous computing systems, i.e. systems containing such components, are becoming increasingly popular in systems ranging from portable mobile devices to high-end supercomputers. Heterogeneous systems are attractive because the specialized computing elements they contain, e.g. GPUs, vector hardware, FPGAs, domain-specific accelerators, and ASICs can greatly improve energy efficiency, performance, or both compared to homogeneous systems. A study of a video encoder [3] demonstrates the significant potential for energy efficiency improvements achievable when targeting specialized architectures. Additionally, domain specific accelerators enable additional performance or energy benefits by exploiting domain specific knowledge [4].

The term "heterogeneous system" is very broad, and its exact meaning should be interpreted based on the context and the time of usage. For example, it may refer to a system including components that differ in micro-architecture, instruction set architecture (ISA), and execution model, such as a combination of a CPU, a GPU and/or special purpose accelerators, or simply refer to a system with cores with different architectural capabilities but same ISA and execution model, such as ARM's big.LITTLE [5]. At the time of writing of this document, the term "heterogeneous system" commonly refers to systems with general purpose CPUs and GPUs, that may or may not be integrated on the same circuit. The system may include special purpose accelerators, that may be programmable. CPUs can execute the operating system and all tasks, including computationally-heavy tasks, but low end CPUs are better suited for traditional, serial workloads. GPUs or vector-enabled multi-

core CPUs handle the computationally heavy application code, mostly exploiting the present data parallelism. An example heterogeneous system is the Qualcomm Snapdragon 865 mobile System-on-Chip (SoC) [6]. It includes a Kryo 585 CPU [7], an Adreno 650 GPU [7], and accelerators, including a programmable Hexagon DSP [8] and special purpose accelerators for audio/video encoding and decoding.

Heterogeneous systems are widely used in a variety of systems, ranging from high-end supercomputers [9, 10, 11], to data centers [12], to mid-level GPU-based systems, to mobile devices and Internet of Things (IoT) devices at the edge of the network. These classes of systems utilize their compute capabilities for different kinds of applications, for example simulations of complex models in several domains (physics, biology, etc) in supercomputers, information retrieval in data centers, image processing in mid-level GPU-based systems, and a variety of applications such as face and speech recognition, computer vision, signal processing [13, 14, 15] depending on the mobile or IoT device. Computing at the edge is becoming increasingly important given the need for real-time data processing at the edge [16]. Edge computing complements cloud computing by doing some computations near the edge of the network, as a means to reduce the volume of transferred data and provide low latency computation. It is estimated that due to devices connected to the Internet that make up IoT, including the sensors and cameras, the amount of data generated will continue to grow, reaching 79.4 zettabytes of data in 2025 [17], increasing the need for computation near data. Therefore, heterogeneous systems on edge devices are an extremely important class of heterogeneous systems, and we expect this trend to continue.

On some of these classes of systems, the application domains they are utilized for are inherently error tolerant, i.e. small errors only introduce acceptable loss in the final output. This observation has led to the development of *approximate computing techniques*, novel hardware architectures and software optimizations that trade-off accuracy for gains in performance and energy [18].

1.1.1 Scope of Heterogeneous Systems for this Thesis

For the scope of this thesis, we will consider heterogeneous systems with multicore CPUs with vector extensions, GPUs, and programmable accelerators. As a use case, we will target PROMISE [19], a programmable mixed-signal Machine Learning (ML) accelerator that achieves performance and energy benefits by performing a wide range of vector dot-product operations in the analog domain.

1.2 PROGRAMMABILITY CHALLENGES OF HETEROGENEOUS SYSTEMS

The wide utilization of heterogeneous systems has given rise to numerous programmability challenges. At a fundamental level, we believe these challenges arise from four root causes. Section 1.2.1 discusses the identified root causes, and section 1.2.2 the resulting programmability challenges.

1.2.1 Root Causes

Diverse hardware parallelism models Different hardware targets expose different, and sometimes multiple, parallelism models. Studying various commonly used hardware targets, we categorize them into the broad hardware classes listed in Table 1.1, based on the parallelism model they expose. Table 1.1 includes a brief description of the exposed parallelism model.

In a heterogeneous system consisting of a combination of the above components, multiple parallelism models may combine in synchronous or asynchronous execution, further increasing the complexity.

Diverse memory systems Differences in parallelism models naturally lead to differences in the underlying memory system. The identified hardware components above have memory systems ranging between cache-coherent memory hierarchies of various depths (two to three levels are common cases), scalar and/or vector register files, scratchpad memory utilized as private or local memory, SRAM blocks (block RAM), and other custom memory designs specialized to custom accelerators. Knowledge of the underlying memory system influences algorithm design and application developing.

The complexity increases further as hardware evolves and offers more parameterizable options to affect the performance, e.g., newer NVIDIA GPU architectures (as of NVIDIA Fermi [20]) provide 64KB scratchpad memory that is configurable (in certain partitions) between L1 cache and local memory, and NVIDIA Volta [21] further increases the allotted scratchpad to 128KB, configurable up to 96 KB of local memory.

Diverse hardware instruction sets Different hardware targets have diverse hardware instruction sets, which translates to different execution semantics and performance characteristics. The diversity in these features makes it especially difficult to achieve object-code portability.

Hardware Class	Parallelism Model	Description
Multicore CPUs	General Multithreading	Multiple threads can execute concurrently in the context of the same process, sharing system resources.
GPUs	Data Parallelism	Single Instruction Multiple Data (SIMD), in which multiple processors execute the same instructions on different pieces of data, with aspects of Single Program Multiple Data (SPMD), since instructions are not necessarily executed at the same time, and the threads executing them have limited communication and synchronization options.
Vector Hardware	(Short) Vector Parallelism	Vector Hardware provides vector lanes of certain width, that can perform the same operation on multiple data points simultaneously.
FGPAs	Spatial Parallelism	A combination of Single Instruction Multiple Thread (SIMT), pipeline and/or dataflow parallelism. Pipeline parallelism is expressed by loop unrolling and buffering. SIMT parallelism is expressed by replication. Dataflow parallelism is expressed using systolic arrays.
Custom Accelerators	Various parallelism forms	Custom to specific accelerators.

Table 1.1: Hardware Classes

Diverse hardware approximation techniques There are multiple approximation techniques at the hardware level, that trade off accuracy at the application level for performance or energy benefits. Depending on the nature of each approximation technique, the "approximation knobs", i.e. there exist (one or more) system-level tuning parameters that affect the application end-to-end behavior and need to be tuned to achieve the programmer intended application specification. The problem is further emphasized when approximation techniques are combined, and different approximation knobs must be tuned to achieve a certain application specification.

1.2.2 Programmability Challenges

Due to the diversities discussed above, programming heterogeneous systems is extremely challenging at multiple levels: parallel language designers, algorithm designers, application developers, compiler developers and hardware engineers must all reason about performance, scalability, and portability across many different combinations of diverse parallel hardware. We discuss the challenges that occur across all these different levels below.

Parallel Language Design It is difficult to design source-level languages to effectively program heterogeneous systems. For example, languages such as CUDA and OpenCL expose data parallelism through a kernel function that is replicated across the data that are to be processed. OpenCL also offers support for dataflow parallelism, targeting FPGAs, albeit through a union and not though a common set of interfaces. Other parallelism models are not specifically supported. Similarly, Streamit [22] supports streaming parallelism. Lime [23] supports dataflow parallelism. We observe that existing source languages only support one or two parallelism models. This creates a steep learning curve for the full utilization of each unique hardware combination in a heterogeneous system, and a prohibitively expensive barrier to overcome for the wide adaption of heterogeneous systems.

Algorithm Design Given the fact that the target hardware affects the exposed parallelism model and underlying memory model, it would be difficult to design a single algorithm, and thus develop a single version of an application, that would perform well across a wide range of diverse parallelism models with their associated underlying memory models. This issue is partially addressed by designing an algorithm that achieves good but not optimal performance or an algorithm that makes the common case fast, or by having multiple versions of an algorithm and selecting the optimal version when the target hardware and the required tuning parameters is known (install time, load time, or run time) [24]. We believe that a combination of the two would be needed to capture a wide range of heterogeneous parallel hardware, as it would be exceedingly difficult to provide optimal implementations for every algorithm for every available hardware target.

Application Development A programmer targeting a heterogeneous system is required to program each hardware component separately, in an appropriate language. This step requires a deep understanding of which application component needs to be mapped to which system hardware component, since the cost of reevaluating a wrong decision is significant.

Source Code Portability Due to the differences in targeted hardware components, any application code implementing a certain operation is subject to the algorithm design challenge. Additionally, the task of compiling source code for an application component down to the hardware component that the respective algorithm is expected to be executed on is complicated due to the vast differences across heterogeneous systems, identified as root causes. Source code portability becomes exceedingly difficult due to the combination of these two factors.

Object Code Portability Heterogeneous systems include different combinations of hardware

- across different vendors of a hardware type, e.g., NVIDIA and ARM GPU.
- across a vendor's family of devices, e.g., different generations or models of NVIDIA GPUs.

Object code portability is an optional, yet important goal for applications that are designed to be able to be executed on a wide range of target platforms that include varying combinations of diverse hardware components. On the contrary, object code portability would not be required for an application that is developed targeting a predetermined system with an ahead-of-time known hardware configuration, e.g. a chemistry simulation developed and optimized for the hardware of the supercomputer scheduled to execute it.

Object code portability is important across devices of the same, and different vendors' devices as well. Vendors must be able to ship a single object code for a broad range of target devices; this way, the same application package can be used to support all devices within that range. This is challenging due to the differences of the underlying hardware targets. The entire stack, including performance analysis and tuning tools, debugging tools, profiling tools, must support the entire (or at least, a large percentage of the) range of the available hardware, both within a single system is concerned, as well as the range of existing system configurations. This stack must be developed and maintained by vendors for each new hardware target and across generations, which can quickly become prohibitively expensive.

Modern apps for mobile phones have used partial solutions (JVM and LLVM virtual ISAs for host CPUs on Apple and Android mobile devices respectively), and workarounds (e.g., custom native libraries for accelerators in the mobile SoCs for Android) to achieve object code portability on mobile devices. However, utilizing the specified hardware components through API calls - that are opaque - makes the offloaded operations not amenable to compiler optimizations along with the rest of the program. The APIs are not retargetable to

other accelerators, thus do not offer benefits should the target accelerator not be available on the target platform. Finally, the native library calls must be expressed explicitly by the programmer, a difficult task considering the number of different accelerators.

Performance Tuning Performance tuning becomes increasingly difficult as the number of tunable parameters increases. As the different hardware components expose different memory hierarchies and parallelism models, the resulting performance models are more complex and less well understood. Additionally, as the number of different hardware components increases, performance tuning becomes prohibitively expensive.

Approximation Tuning On top of the hardware approximation techniques, multiple software approximation techniques also become available to the programmer as tools to trade off accuracy at the application level for performance or energy benefits. Similar to hardware techniques, software approximations also need to expose approximation knobs, software- or application-level, that affect the application end-to-end behavior. For a combination of hardware and software techniques, the pool of approximation knobs that the programmer is responsible for understanding and tuning may also be at different levels, application-level, software-level, and system-level. and deep understanding of all software and hardware approximation techniques, as well as the effect of their interactions is required. This is extremely difficult, and also depending on the approximation technique or combination thereof may not be possible.

The programmability challenges described above also emerge, as expected, on heterogeneous systems used for demanding computation tasks on edge computing, with the effect of further limiting the computational power on these already resource constrained systems. For example, deep learning inference tasks executing on edge devices are generally unable to fully utilize the available hardware. Instead application programmers tend to fall back to targeting the lowest common denominator, the heterogeneous system's core processor. For example, inference tasks on the Facebook app utilize general, algorithmic optimizations that can target all processing environments for performance benefits, opting to use of accelerators only when there is little diversity or there is control over the system environment using virtual reality platforms [25]. Given the diversity of heterogeneous SoCs in mobile devices, this problem is extremely difficult to overcome: there is no standard mobile SoC to optimize for, and target hardware is extremely fragmented [25].

1.3 SUMMARY OF RELATED WORK

This section briefly discusses the compiler infrastructure that aims to address the programmability challenges of heterogeneous systems. We also discuss systems developed in the domain of approximate computing, since it is extremely relevant in applications domains whose computation is allocated to heterogeneous systems. A detailed literature review is provided in chapter 2.

1.3.1 Virtual ISAs

An approach to address the programmability challenges of heterogeneous systems is to abstract the diversities that we identified as their root causes under a virtual layer. This layer effectively presents a hardware abstraction that is a uniform representation of the underlying hardware and is targeted by the layers higher on the software stack. Applications can be developed, compiled, and shipped targeting the instruction set of the virtual hardware, or virtual instruction set (Virtual ISA). Dedicated translators, or compiler backends, are responsible for translating from the Virtual ISA to the native ISA for execution on a supported device within the target family at install time or runtime, thus achieving portability of "virtual object code" across the corresponding family of devices. Some widely used heterogeneous systems define virtual instruction sets spanning one or more families of devices, e.g., PTX for NVIDIA GPUs, HSAIL for CPUs and GPUs from several vendors and SPIR-V for devices running OpenCL.

Depending on the challenges that a virtual ISA is designed to solve, it abstracts away the relevant differences in the underlying hardware. For example, these virtual ISAs abstract away the differences in underlying ISAs, but not in parallelism models. Except for SPIR-V, which is essentially a lower-level representation of OpenCL for the compute kernels of the OpenCL device programs, these virtual ISAs are primarily focused on GPUs and do not specifically address other hardware classes, like vector hardware or FPGAs. That is due to the parallelism model they expose, which is primarily a "grid of kernel functions". Moreover, *none of these virtual ISAs* address the challenges of compiler optimizations through the use of a compiler IR or serve as the basis of runtime scheduling.

1.3.2 Parallel Compiler IRs

Previous parallel compiler IR we know of (for example, [26, 27, 28, 29, 30]) do not aim to, or have different limitations in, targeting heterogeneous parallel systems, as described in

more detail in section 2.

1.3.3 Approximate Computing Frameworks

Many approximation techniques have been proposed, to improve performance or energy efficiency. Our work draws from the pool of approximation techniques, and focuses on managing them in a single framework for adaptive approximation tuning.

Existing systems for accuracy-aware optimizations do not provide a fully automated framework that is able to target multiple heterogeneous devices with diverse approximation choices without requiring programmer-guided low-level annotations. The ACCEPT [31] framework optimizes program computations given a source-level end-to-end quality metric, and can target diverse compute units. However, it does not introduce approximation information at the IR or virtual ISA level, which limits the approach to the specific source-level annotations used. It also does not attempt to provide software portability. EnerJ [32] presents a type system that separates approximate and precise data. Rely [33] and Chisel [34] introduced the idea of quantifiable reliability and accuracy at the program level. However, introducing approximation metrics as part of a new programming language is not conducive with our goal of program portability at the object code level, since applications need to be ported at the source-level.

Additionally, all these systems require programmer annotations at individual function level or lower to drive the approximation selection.

There also exist approximation driven adaptive systems [35, 36, 37, 38, 39, 40], that utilize one or a limited number of approximation techniques, to dynamically tune the approximation level utilized by an application in response to power or load variations. These systems, which we will describe in more detail in section 2, do not take into account object code portability, and they do not target heterogeneous hardware with diverse approximation opportunities.

1.4 APPROACH

To address the programmability challenges of heterogeneous systems, we identify a strategy that involves a three-part approach.

As a first step, we target heterogeneous systems that include hardware components that expose different parallelism models, memory systems and hardware ISAs. Section 1.4.1 summarizes our approach to enhance the object code and performance portability of such systems. This work was done jointly, and equally, with Prakalp Srivastava (prakalps@google.com), and appears in both our theses.

As a second step, we focus on heterogeneous systems that additionally include accelerators that expose approximation options. Section 1.4.2 describes the approach to exploit the performance and energy efficiency superiority of heterogeneous systems by extending our prior work for accuracy-aware optimizations. This work is lead by Hashim Sharif, and will be included in the theses of Hashim Sharif (hsharif3@illinois.edu) and Prakalp Srivastava (prakalps@google.com).

As a final step, we extend to heterogeneous systems at the edge. Section 1.4.3 describes our approach to address the challenges of performing computationally demanding tasks on the constrained edge devices, often under changing conditions. This work is done jointly, and equally, with Hashim Sharif (hsharif3@illinois.edu) and will be included in both our theses.

1.4.1 Heterogeneous Parallel Systems

We believe that the object code and performance portability challenges of heterogeneous systems can be best addressed by developing *a single parallel program representation flexible enough to support at least three different purposes*:

1. A *compiler intermediate representation*, for compiler optimizations and code generation for diverse heterogeneous hardware. Such a compiler IR must be able to implement a wide range of different parallel languages, including general-purpose ones like OpenMP, CUDA and OpenCL, and domain-specific ones like Halide and TensorFlow.

2. A *virtual ISA*, to allow virtual object code to be shipped and then translated down to native code for different heterogeneous system configurations. This requirement is essential to enable application teams to develop and ship application code for multiple devices within a family.

3. A *representation for runtime scheduling*, to enable flexible mapping and load-balancing policies, in order to accommodate static variations among different compute kernels and dynamic variations due to external effects like energy fluctuations or job arrivals and departures.

We believe that a representation that can support all these three capabilities could (in future) also alleviate other programmability challenges of heterogeneous systems, specifically simplify parallel algorithm development and influence parallel language design, although we do not explore those in this work.

In this work, we propose such a parallel program representation, Heterogeneous Parallel Virtual Machine (HPVM), and evaluate it for three classes of parallel hardware: GPUs, SIMD vector instructions, and multicore CPUs. Our evaluation shows that HPVM can serve all three purposes listed above: a compiler IR, a virtual ISA, and a scheduling representation, as described below.

The parallel program representation we propose is a *hierarchical dataflow graph with shared memory*. The graph nodes can represent either coarse-grain or fine-grain computational tasks. In this work, we focus on moderately coarse-grain tasks (such as an entire inner-loop iteration). The dataflow graph edges capture explicit data transfers between nodes, while ordinary load and store instructions express implicit communication via shared memory. The graph is hierarchical because a node may contain another dataflow graph. The leaf nodes contain scalar computations. A graph node represents a static computation, and any such node can be "instantiated" in a rectangular grid of dynamic node instances, representing parallel instances of the computation. In this case, the incident edges of the node are instantiated as well, in order to capture the connections between the dynamic instances of the source and sink nodes.

The hierarchical dataflow graphs naturally capture all the important kinds of coarse- and fine-grain data and task parallelism in heterogeneous systems. In particular, the graph structure captures coarse-grain task parallelism (including pipelined parallelism in streaming computations); the graph hierarchy captures multiple levels and granularities of nested parallelism; and the node instantiation mechanism captures either coarse- or fine-grain SPMD-style data parallelism.

We describe a prototype system (also called HPVM) that supports all three capabilities listed earlier. The system defines a compiler IR as an extension of the LLVM IR [41] by adding HPVM abstractions as a higher-level layer describing the parallel structure of a program.

As an example of the use of HPVM as a compiler IR, we have implemented an illustrative compiler optimization, *graph node fusion*,which operates directly on the HPVM dataflow graphs.

Node fusion achieves "kernel fusion", and the graph structure makes it explicit when it is safe to fuse two or more nodes. Similarly, we find that the graph hierarchy is also an effective and *portable* method to capture tiling of computations, which can be mapped either to a cache hierarchy or to explicit local memories such as the scratchpads in a GPU.

To show the use of HPVM as a virtual ISA, we implemented *translators* for NVIDIA GPUs (using PTX), Intel's AVX vector instructions, and multicore X86-64 host processors using Posix threads. The system can translate *each HPVM graph node* to one or more of

these distinct target architectures (e.g., a 6-node pipeline can generate $3^6 = 729$ distinct code configurations from a single HPVM version). Experimental comparisons against hand-coded OpenCL programs compiled with native (commercial) OpenCL compilers show that the code generated by HPVM is within 22% of hand-tuned OpenCL on a GPU (in fact, nearly identical in all but one case), and within 7% of the hand-tuned OpenCL in all but one case on AVX.

Finally, to show the use of HPVM as a basis for runtime scheduling, we developed a graph-based scheduling framework that can apply a wide range of static and dynamic scheduling policies that take full advantage of the ability to generate different versions of code for each node. Although developing effective scheduling policies is outside the scope of this work, our experiments show that HPVM enables flexible scheduling policies that can take advantage of a wide range of static and dynamic information, and these policies are easy to implement directly on the HPVM representation.

1.4.2 A Compiler for Approximate Heterogeneous Parallel Hardware

This phase of our work extends to heterogeneous parallel systems with approximate hardware, in order to utilize approximate computing techniques (section 1.3) to achieve performance or energy benefits.

A given computational algorithm or kernel may benefit from multiple different approximation techniques, and moreover, a realistic application will contain several (or many) distinct kernels. Determining how best to map such an application to a modern heterogeneous system while achieving the best overall tradeoff between end-to-end application-level accuracy and performance or energy is an open research problem. Moreover, section 1.2 described the *Approximation Tuning* programming challenge. In the context of heterogeneous approximate hardware, this expresses itself as the fact that application developers and end users cannot be expected to specify error tolerances in terms of the system-level approximation knobs required by the various hardware approximation techniques, or even know about many of them. To target heterogeneous systems with approximate hardware we need automated mapping strategies that can translate application-level specifications (e.g., tolerable classification error in a machine learning application) to system-level knobs (e.g., neural network parameter precision or circuit-level voltage swings).

Optionally, for modern applications targeting a wide range of diverse systems, software portability is a requirement, not just at the source-code level but also the ability to *ship* software that can execute efficiently (as described in section 1.2.2). Modern applications for both desktop and mobile (e.g., smartphone or tablet) systems are almost always shipped by

application teams to end-users in a form that can execute on multiple system configurations (e.g., with different vector ISAs or GPUs) and even multiple hardware generations (e.g., across X86 processors). GPUs, for example, provide virtual instructions sets, e.g., PTX or HSAIL, to enable software to be shipped as "virtual object code" that is translated to particular hardware instances only on the end-user's system. This is a major challenge for approximate computing approaches because hardware-specific accuracy-performance-energy choices can make orders-of-magnitude difference in the performance and energy benefits achieved in exchange for relaxing accuracy. An important goal for real-world use of such approaches is to enable software to be shipped as *portable* virtual object code, while deferring the hardware-specific aspects of accuracy-performance-energy optimizations to be performed after shipping [42] (e.g., on the end-user's device or on servers in an "application store").

In order to defer hardware-specific optimizations to install-time (or later), an appropriate virtual object code representation must include the necessary or relevant information to inform the decision. For accuracy-aware optimizations to benefit from a virtual object code representation, approximation information must be included at the virtual object code.

To that end, we extend HPVM to ApproxHPVM, a compiler IR and framework for accuracy-aware optimizations. ApproxHPVM automatically identifies approximable operations, by allocating a user provided, high level and end to end accuracy metric, to operation-specific error budgets, and maps the identified approximable operations to fast, approximate hardware to achieve performance and energy benefits.

The ApproxHPVM IR extends the HPVM IR in two key ways, by including:

- domain-specific knowledge by incorporating high-level tensor operations as part of the IR.
- hardware-independent approximation metrics that quantify the acceptable accuracy loss for individual operations.

As ApproxHPVM does not incorporate any hardware-specific knowledge as part of the IR, it maintains the HPVM rationale of serving as a fully self-contained, hardware-agnostic virtual ISA that can be shipped and mapped to a variety of hardware platforms. This is enabled by partitioning the accuracy-energy-performance optimizations into a hardware-agnostic stage and a hardware-specific stage, where software can be shipped between the two stages. The approximation metrics are computed as part of the hardware-agnostic stage, and shipped as part of the virtual ISA, enabling object code portability for approximate application code.

The HPVM compiler has been extended with backends for two hardware targets for ApproxHPVM, PROMISE [19] , and NVIDIA GPUs through an optimized cuDNN/cuBLAS

runtime library, and with support for the ApproxHPVM high level tensor operations in the HPVM C/C++ frontend. ApproxHPVM programs can also be written in Keras [43], a popular neural-network library that can run on top of TensorFlow and Theano among other frameworks. Keras provides a simple, programmable interface for providing high-level descriptions of neural networks. We provide a Keras frontend that automatically generates the virtual ISA for ApproxHPVM programs.

Experimental results show significant gains, up to 9x performance speedups and up to 11.3x energy reduction, while staying within user-specified application-level quality metrics with high probability, for nine deep learning benchmarks and five convolution-based image processing benchmarks.

1.4.3 A Compiler and Runtime System for Adaptive Approximations

The final phase of our work specifically targets heterogeneous systems at the edge.

We identify the following characteristics of these systems, whose combination is important for effectively targeting heterogeneous systems at the edge.

- Highly diverse and resource constrained: Section 1.2 discusses the programmability challenges of heterogeneous systems, and how they effectively limit the computational power of the already resource constrained edge devices.

- Amenable to approximation: The application domains often processed on such systems are inherently error tolerant, i.e. they can tolerate small amounts of error with acceptable resulting quality. This is extremely valuable when targeting edge systems, since the specific applications in these domains are highly compute-intensive, and may be far beyond the capabilities of even near- and medium-future edge compute hardware. Edge compute hardware, even with custom accelerators, is often too limited in computational power, memory capacity, or energy capacity to support these computations.

- Varying conditions: When targeting edge devices, the varying conditions under which an application might be executing create additional complexities. Load variations, potentially caused by the need to execute different number of applications or an application under different quality requirements, and power constraints caused by battery, may alter the requirements for energy efficient execution of an application. These variations can occur unpredictably in the uncontrolled environment of edge computing.

Thus, to effectively target heterogeneous systems at the edge, we require a framework that abstracts away the diversity of the underlying hardware, enables energy efficient computation

through approximation optimizations, and provides an adaptive control mechanism to tune the approximation level in order to alter the achieved performance or energy benefits.

To that end, we propose ApproxTuner (an extension of ApproxHPVM), a portable optimizing compiler and runtime system that targets heterogeneous systems at the edge.

In order to address the programmability challenges, ApproxTuner is built on top of ApproxHPVM, and thus utilizes a portable virtual ISA and compiler.

We identify tensor operations representative of the image processing domain and extend the ApproxHPVM IR and compiler infrastructure for the image processing domain. The framework enables the existing general purpose approximation techniques to be seamlessly used for the image processing tensor intrinsics, and we additionally implement an approximation technique applicable to the image processing intrinsics.

ApproxTuner provides *several software and hardware approximations* for tensor operations used in two different domains: convolutional neural networks and image processing. Moreover, the framework is extensible to add new approximation techniques.

Although a wide range of approximation techniques is important for targeting heterogeneous systems at the edge, supporting several different approximations is in conflict with portability. This is because (a) autotuning is very expensive because the combined search space is large, but (b) ahead-of-time hardware-agnostic autotuning for the approximations is not conducive to performance portability since approximation choices are highly hardware-dependent.

Additionally, we find that software approximations present difficulties for approximation tuning. We identify that the accuracy impact of software approximations in ApproxTuner cannot be easily captured with simple attributes (e.g., L1 or L2 norms of output tensors, which are the error metrics in the IR). In particular, we observe that L1 and L2 norms work well when approximation behavior can be captured by independent random variation, such as external noise (e.g. PROMISE, low-voltage circuits, random bit flips). However, software approximations often display input sensitivity, thus their effect does not display random variation. We observe that the L1-L2 approximation metrics do not capture their effect on the computation (section 7.1). Thus combining software and hardware approximations in ApproxTuner cannot be achieved using the approximation parameters at the IR level, as approached by the ApproxHPVM work.

The ApproxTuner system addresses these challenges by decomposing the approximation tuning in three carefully designed steps: *development-time*, *install-time*, and *dynamic* adaptive approximation tuning. Development-time tuning uses an autotuner to tune the available hardware-independent approximations, and construct *approximate* Pareto-optimal curves for their parameter values. Install-time tuning measures the actual performance and energy of

the points in the approximate Pareto curve and updates the curve to reflect these measured values. For hardware-specific approximations (which are inherently non-portable), it also uses an autotuner to select and tune among the full set of software and hardware approximations. The install-time stage produces an accurate Pareto-optimal curve for the particular hardware, in order to enable efficient dynamic tuning. Finally, at run time, approximation choices are periodically updated in response to changing resource variations, e.g., system load, battery capacity, or task deadlines, using the Pareto-optimal curve to select approximation choices and parameter settings for each tensor operation.

The three step tuning strategy enables *performance portability of application packages*, while supporting both software and hardware approximation techniques. Performance portability is a critical requirement for modern applications, since applications are expected to run efficiently on a wide range of systems. The shipped application package includes hardware-independent approximation methods, therefore they are applicable to any hardware target, and may be further refined at install time with install time tuning.

The install-time retuning phase can be prohibitively expensive on resource-constrained edge devices. To render the retuning feasible, we propose *federated autotuning*. The federated autotuning process requires an interaction between a centralized server (e.g., an app store) and client devices, where each client performs a small fraction of the autotuning task and shares the partial results with the centralized server. The server caches the results gathered from the client devices and appropriately updates the Pareto curve of configurations for the target device. In this manner, each edge device only incurs a small fraction of the cost of install-time autotuning. Carefully designed search space pruning is also utilized to speed up install time tuning.

We evaluate ApproxTuner using benchmarks from the domains of machine learning and image processing. For 7 Deep Neural Network (DNN) benchmarks, we find that ApproxTuner achieves geometric mean speedup of 1.9x and geometric mean energy reduction of 2x with only hardware-independent approximations. We also show that install-time tuning, which enables mapping to a machine learning accelerator, PROMISE, can improve these results to geometric mean speedup of 5.6x and geometric mean energy reduction of 5.9x. Similarly, for three image processing benchmarks, we achieve a geometric mean speedup of 2.14x and a geometric mean energy reduction of 2.4x. Finally, we show the adaptability of ApproxTuner through dynamic approximation tuning, which for our DNN and image processing benchmarks can deliver required speedups by varying performance and accuracy at runtime.

1.5 CONTRIBUTIONS

We summarize the contributions of this thesis, as a result of the approach described in 1.4.

Heterogeneous Parallel Systems I[1] consider the object code and performance portability challenges of heterogeneous parallel systems.

- I develop a parallel program representation (HPVM) for heterogeneous parallel systems based on a hierarchical dataflow graph with side effects, which captures essentially all the important kinds of task- and data-parallelism on heterogeneous systems.
- I implement an HPVM prototype system on top of a widely used compiler infrastructure, LLVM, which historically has lacked any *explicit* support for heterogeneous parallel systems in the LLVM IR.
- I show that HPVM can be used to create an effective parallel virtual ISA for heterogeneous systems by (a) using HPVM as a *persistent* representation of programs, and (b) by implementing translators from HPVM to three different classes of parallel hardware: GPUs, vector SIMD, and multicore CPUs.
- I show that HPVM can be used as an effective parallel compiler IR, that can support important optimizations like node fusion and tiling.
- I show that HPVM dataflow graphs can be used to support flexible static and dynamic scheduling policies, that take full advantage of the ability to translate individual HPVM graph nodes to multiple hardware.

Heterogeneous Parallel Systems with Approximate Hardware I[2] extend to heterogeneous parallel systems with approximate hardware.

- I extend the HPVM compiler infrastructure with high-level tensor operations plus hardware-agnostic approximation metrics that quantify the accuracy of unreliable and approximate computations. I have shown that ApproxHPVM IR can serve as a fully

[1]This work was done jointly, and equally, with Prakalp Srivastava (prakalps@google.com), and appears in both our theses. We contributed equally to the design of the HPVM parallel program representation abstractions, implementation of HPVM prototype system on top of LLVM, and code generation. I lead the aspects of HPVM as a compiler IR and runtime representation.

[2]This work is lead by Hashim Sharif, and will be included in the thesis of Hashim Sharif (hsharif3@illinois.edu) and Prakalp Srivastava (prakalps@google.com). I contributed to the ApproxHPVM IR design, and lead its implementation as part of the HPVM compiler infrastructure.

self-contained virtual ISA, and so software can be shipped to achieve virtual object-code portability for approximable computations and allow deferred accuracy aware optimizations.

- I implement translators from ApproxHPVM to two hardware targets, PROMISE and an optimized cuDNN/cuBLAS runtime library for NVIDIA GPUs, extend the existing C/C++ HPVM frontend with support for the ApproxHPVM intrinsics, and contribute to the Keras frontend.

Heterogeneous Parallel Systems at the Edge Targeting heterogeneous systems at the edge, I[3] make the following contributions:

- Several software and hardware approximations in a single framework: ApproxTuner is extended (from ApproxHPVM) with a range of software approximation techniques, now including both software and hardware approximations. A range of approximation techniques enables better optimization of a single application, as more combinations of the approximation techniques are available to provide accuracy and performance-energy benefits.

- Three-phase approximation tuning: I propose a novel three-phase approach to approximation tuning that provides performance portability, retargetability to different compute units (with varying hardware-specific approximation knobs), and dynamic approximation tuning, by splitting approximation tuning into: a) selection of hardware-independent approximations at *development-time*, b) selecting hardware-specific approximation options at *install-time*, and c) a fast approximation selection at *runtime*.

- Evaluation: ApproxTuner evaluation shows significant performance and energy benefits through hardware-independent approximations, that are significantly improved with the use of hardware-specific approximation techniques after install-time retuning. We also show that dynamic approximation tuning can deliver required speedups by varying performance and accuracy at runtime.

[3]This work is done jointly, and equally, with Hashim Sharif, and will also appear in his thesis. We contribute equally to the approximation techniques utilized in the ApproxTuner system and the three-step approximation tuning design. I lead the compiler implementation and runtime approximation tuning. Hashim lead the autotuning in general, specifically the install time tuning with the federated autotuning and other techniques to make expensive install-time autotuning practical. We both contribute to development time tuning.

1.6 THESIS ORGANIZATION

The rest of this document is organized in the following chapters: Chapter 2 gives context to this thesis by presenting an overview of the state of the art in the relevant domains. Chapter 3 presents the HPVM Intermediate Representation design, that aims to address the object code and performance portability challenges of heterogeneous systems. Chapter 4 presents the HPVM prototype implementation based on the HPVM IR design. The prototype system is a proof of concept, to show that the design in fact addresses the portability challenges of heterogeneous systems, and includes the implementations of HPVM as a virtual ISA, use of HPVM as a compiler IR, and as a basis for runtime scheduling. Chapter 5 includes the evaluation of the HPVM system in respect to the above. Chapter 6 presents ApproxHPVM, an extension to HPVM for approximation information at the IR level. Chapter 7 presents ApproxTuner, that addresses the programmability challenges of heterogeneous systems at the edge. Finally, Chapter 8 presents a summary of the contributions of this thesis and future directions of this work.

CHAPTER 2: RELATED WORK

This chapter provides a literature review of the state of the art in programming languages, compiler infrastructures, and runtime systems that aim to address the programmability challenges of heterogeneous systems, and techniques and systems developed in the emerging domain of approximate computing.

2.1 PROGRAMMING TECHNOLOGIES FOR HETEROGENEOUS SYSTEMS

There is a long history of work on dataflow execution models, programming languages, and compiler systems for homogeneous parallel systems [44, 45, 46, 47, 30, 48, 49, 50]. HPVM aims to adapt the dataflow model to modern heterogeneous parallel hardware. We focus below on programming technologies for heterogeneous systems.

Virtual ISAs: NVIDIA's PTX virtual ISA provides portability across NVIDIA GPUs of different sizes and generations. HSAIL [51] and SPIR-V [52] both provide a portable object code distribution format for a wider class of heterogeneous systems, including GPUs, vectors and CPUs. All these systems implement a model that can be described as a "grid of kernel functions", which captures individual parallel loop nests well, but more complex parallel structures are only expressed via explicit, low-level data movement and kernel coordination. This makes the underlying model unsuitable for use as a retargetable compiler IR, or for flexible runtime scheduling. Finally, it is difficult, at best, to express some important kinds of parallelism, such as pipelined parallelism (important for streaming applications or other accelerators), because all buffering, synchronization, etc., must be implemented explicitly by the program. Specifically, while the parallel pipeline stages can be expressed using different command queues, synchronizing every data transfer between every pair of pipeline stages requires buffer management, specified by the programmer, and use of explicit "events". An *event* encodes an ordering between two operations, and is consumed upon occurrence. Therefore, a different event would be needed to be enqueued to capture all dependencies required to represent a pipeline per data transfer per stage.

In contrast, pipelined parallelism can be expressed easily and succinctly in HPVM (in addition to coarse- or fine-grain data-parallelism). Pipeline parallelism is captured naturally as part of the HPVM representation, and the implementation details such as buffering and synchronization are left to be handled by the system.

Compiler IRs with Explicit Parallel Representations: We focus on parallel compilers for heterogeneous systems. The closest relevant compiler work is OSCAR [26, 53, 27],

which uses a hierarchical task dependence graph as a parallel program representation for their compiler IR. They do *not* use this representation as a virtual ISA, which means they cannot provide object code portability. Their graph edges represent data and control dependencies, *not dataflow* (despite the name), which is well suited to shared memory systems but not as informative for non-shared memory. In particular, for explicit data transfers, the compiler must infer automatically what data must be moved (e.g., host memory to accelerator memory). They use hierarchical graphs only for the (homogeneous) host processors, not for accelerators, because they do not aim to perform parallel compiler transformations for code running within an accelerator nor runtime scheduling choices for such code. KIMBLE [54, 28] adds a hierarchical parallel program representation to GCC, while SPIRE [29] defines a methodology for sequential to parallel IR extension. Neither KIMBLE nor SPIRE make any claim to, or give evidence of, performance portability or parallel object code portability.

Delite Domain Specific Languages (DSLs) use the Delite compiler framework (described below) to define an IR for the new DSL, The IR is a sea of nodes representation; each node explicitly specifies its data and control dependencies, but otherwise is free to float. Nodes are simultaneously viewed in multiple ways: a generic layer representing a definition, a parallel layer called Delite *ops*, and a domain specific layer that contains the actual operation, e.g. `VectorPlus`. The IR is optimized by viewing nodes at any layer, allowing for generic and domain specific operations to be applied at the same IR. The generated IR is domain specific and not general purpose like HPVM. However, the feature of simultaneous view of Delite IR in multiple ways would be useful as we extend the HPVM IR with domain specific information for the second phase of our work, in order to seamlessly allow both domain independent and domain specific optimizations.

HeteroIR [55] is an intermediate language that used to map high level programming models (e.g. OpenACC, with OpenARC [56]) to diverse heterogeneous devices. HeteroIR encapsulates the common accelerator operations in high level function calls, and during application execution the calls are dispatched to the appropriate target architecture API by the runtime system. However, the HeteroIR constructs only capture the host part of the program, the accelerator kernels are not part of the HeteroIR representation. In contrast, HPVM captures the computation and data transfers as part of the dataflow graph, enabling analysis and potentially mapping to hardware components at different granularity.

AOMP [57] is AMD's compiler that supports OpenMP and offloading to multiple GPU targets. It utilizes an LLVM-based IR, that represents OpenMP specific information instead of abstracting the parallel structure of the program using a uniform parallelism model.

Compiler Frameworks for Heterogeneous Systems: Habanero-Java [58] and Habane-

ro-C [59], provide an abstraction of heterogeneous systems called *Hierarchical Place Trees* (HPT), which can be used to express and support flexible mapping of parallel programs. The HPT model supports co-allocation of data and computation at multiple levels of a memory hierarchy. It aims to address the diversity in memory systems of heterogeneous systems, but does not address the diversity in underlying instruction sets, therefore not addressing portability of object code through a representation usable as a portable virtual ISA.

In Tangram [60], a program is written in interchangeable, composable building blocks, called *codelets*. Codelets represent different, semantically equivalent, algorithmic or implementation choices for an application code snippet. Tangram defines a set of transformations that allow for composition of new codelets. The compiler will optimize for the target architecture by selecting or composing good codelets, and tuning the tunable parameters for the target architecture. Exploring algorithmic choices is orthogonal to, and can be combined with, our approach, since HPVM is a lower level representation.

Delite [61] is a compiler framework for developing compiled, embedded DSLs inside the programming language Scala. Delite DSLs construct an IR on which both generic and domain specific optimizations are performed and from which compilation is performed to target hardware. The Delite Execution Graph (DEG) encodes the dependencies between the computations in the program, and kernels are generated implementing the *ops*, or IR nodes, for different targets. Delite provides common parallel patterns and optimizations that can be reused across DSL implementations. To provide flexibility to run these ops on different hardware devices, Delite relies on the DSL developers to provide Scala, CUDA, OpenCL implementations of these computations as necessary for efficiency. HPVM on the other hand relies on the hardware vendors to provide platform specific implementation of computations in HPVM IR. The broader Delite approach can be combined with HPVM approach to ease burden on the DSL developers.

Multi-Level Intermediate Representation (MLIR) [62] is a compiler infrastructure aiming to unify the effort towards compilation for heterogeneous hardware and domain specific languages. The MLIR infrastructure defines an Intermediate Representation based on Static Single Assignment (SSA) form with nested instead of flat regions. It allows the definition of customizable Ops, organized within separate namespaces called *dialects*, representing the set of operations that are legal and supported for a domain. Compiler transformation can be defined within a dialect, or for all dialects when reasonable, e.g., inlining. MLIR Ops can represent different levels of abstraction. Thus, MLIR captures language abstractions at different levels, and operates on the principle of progressive lowering, sequencing compiler transformations to happen at higher abstraction levels before progressively moving to a lower abstraction level, effectively losing the information contained in the higher abstraction level.

HPVM can be implemented as a dialect of MLIR, defining Ops for the HPVM abstractions, and use translators to convert to available Dialects, e.g. GPU, SPIR-V, or other supported dialects with hardware support, thus utilizing the MLIR infrastructure for optimization and code generation for these targets. Approximation information may be introduced as well, as currently MLIR includes no approximation abstractions. Implementation of HPVM as an MLIR dialect would eliminate the need to directly use the dialects supported by the HPVM translators in order to target a range of parallel hardware. MLIR dialects could use HPVM, instead of going through separate individual dialects for each target hardware.

However, it is explicitly a non goal of the MLIR infrastructure to provide the runtime support required for the implemented Dialects, and thus runtime scheduling and approximation tuning in HPVM and ApproxTuner systems would still be explicitly managed by our runtime. Thus, MLIR infrastructure can be used to implement HPVM as a dialect, which can benefit the code generation and optimization options of HPVM, but does not replace the HPVM's capabilities as a Virtual ISA and a runtime representation.

Runtime Libraries for Heterogeneous Systems: Parallel Virtual Machine (PVM) [63] enables a network of diverse machines to be used cooperatively for parallel computation. Despite the similarity in the names, the systems have different goals. What is virtualized in PVM are the application programming interfaces for task management and communication across diverse operating systems, to achieve portability and performance across homogeneous parallel systems. HPVM virtualizes the parallel execution behavior and the parallel hardware ISAs, to enable portability and performance across heterogeneous parallel hardware, including GPUs, vector hardware, and potentially FPGAs.

Several other runtime systems [64, 65, 66, 67] support scheduling and executing parallel programs on heterogeneous parallel systems. However, they do not address the challenge of object code portability.

Programming Languages: Source-level languages such as CUDA, OpenCL, OpenACC, and OpenMP all support a similar programming model that maps well to GPUs and vector parallelism. None of them, however, address object code portability and none can serve as a parallel compiler IR. They also make it difficult to express important kinds of parallelism, like pipelined parallelism. OpenCL also offers support for dataflow parallelism, targeting FPGAs, albeit through a union and not though a common set of interfaces.

PENCIL [68] is a programming language defined as a restricted subset of C99, intended as an implementation language for libraries and a compilation target for DSLs. Its compiler uses the polyhedral model to optimize code and is combined with an auto-tuning framework. It shares the goals of source code portability and performance portability with HPVM. However, it is designed as a readable language with high-level optimization directives rather

than as a compiler IR, per se, and it also does not address object code portability.

StreamIt [22] and CnC [69] are programming languages with a somewhat more general representation for streaming pipelines. For example, StreamIt *filters* are the basic unit of computation, and they can be composed using the following structures: *Pipeline*, *SplitJoin*, and *FeedbackLoop*. FeedbackLoop allows cycles to be introduced in the pipeline representation, while the HPVM dataflow graph is acyclic. They, however, focus on stream parallelism, whereas HPVM supports both streaming and non-streaming parallelism. This is crucial when defining a compiler IR or a virtual ISA for parallel systems (of any kind), because most parallel languages (e.g., OpenMP, OpenCL, CUDA, Chapel, etc.) are used for non-streaming parallel programs.

Legion [70, 71] is a programming model and runtime system for writing high-performance applications for distributed heterogeneous architectures. It provides abstractions for describing the structure of program data in a machine independent way. Tasks can be defined hierarchically, i.e. a Legion program is a tree of tasks with a top level task and tasks can spawn subtasks. Tasks specify logical regions they will operate on, and define the privileges and coherence for each region. The Legion runtime maps logical regions to (one or more instances of) physical memory. Legion defines a memory model that provides control over data placement and partitioning on top of complex memory hierarchies in a machine independent way, which HPVM lacks. However, Legion focuses on data placement due to complex memory hierarchies of distributed heterogeneous memory systems where cost of communication dominates the cost of computation. It does not address the challenges of diversity in parallelism models or instruction sets, as code for leaf tasks (subtasks that do not spawn other tasks) is provided by the programmer for each target that the leaf task is expected to execute on.

Similarly, Sequoia [72, 73] provides rich memory abstractions to enable explicit control over movement and placement of data at all levels of a heterogeneous memory hierarchy. Specifically, machines are abstracted as trees of distinct memory modules defining their memory hierarchy. Sequoia programs abstractly describe hierarchies of tasks, with tasks specifying information about communication and working set of data. Computation is restricted to tasks that will eventually be mapped to the leaf level of the machine tree.

HPVM lacks these features, but does express tiling effectively and portably using the hierarchical graphs. In future, we aim to add richer memory abstractions to HPVM. We believe that HPVM could borrow ideas from Legion and Sequoia for that purpose, as briefly mentioned in section 8.2.

Approximation-Aware Languages. EnerJ [32] presents a type system that separates approximate and precise data. The developer needs to annotate each variable in the source code as precise or approximate before the type system can ensure that the approximate data is never assigned to the precise variable. More recently, Decaf [74] performs type inference to reduce the developer annotation effort.

Rely [33] and Chisel [34] introduced the idea of quantifiable reliability and accuracy at the program level. They define function-level specifications that express the maximum probability with which the function can produce an inaccurate result. These specifications separate the optimization within the function from the uses of the function: the code that calls the function can rely on the specification, while the body of the function can be modified separately and Rely and Chisel can statically verify that those implementations satisfy the specification.

All these systems require programmer annotations at the source level. ApproxHPVM introduces the concept of quantifiable reliability at the IR level. Incorporating approximation metrics at the IR level provides a more portable alternative, since the metrics are preserved even after compiling the program and the approximation becomes a first-class citizen in a compiler workflow, which is able to interact with various front-end languages and hardware-specific features. ApproxTuner decouples software from hardware approximation tuning as part of its three step approximation tuning strategy, enabling hardware independent, development time software approximation options to be shipped as part of the virtual ISA and hardware-specific tuning to occur at install time, as well as dynamic approximation tuning.

Compiler-based Systems for Machine Learning. TVM [75] proposes a compiler framework that supports the compilation and optimization of machine learning workloads on multiple hardware targets. Similarly, Glow [76] and XLA [77] are also ML-based compiler frameworks that leverage DNN-specific operations in the IR design, thereby facilitating domain-specific optimizations. As ApproxHPVM also leverages domain-specific information with the inclusion of high-level tensor intrinsics, it also facilitates such domain-specific optimizations. None of these systems include approximation metrics in the IR design and hence do not provide the portability and flexibility offered by ApproxHPVM. While these existing systems provide support for precision-tuning to half-precision floating point (FP16) and 8-bit integer (INT8), ApproxHPVM provides more extensive support for approximation since it also allows for mapping computations to accelerators that provide performance-energy-accuracy trade-offs. Moreover, ApproxHPVM enables approximation mechanisms

that are not limited to the machine learning domain. Additional approximation techniques are introduced in ApproxTuner.

Autotuning Systems. The PetaBricks programming language automatically autotunes the choice of an algorithm among multiple user-provided choices with varying accuracy and performance characteristics [24, 42, 78, 79]. Its auto-tuner uses heuristic algorithms to search among alternative program implementations. Autotuning algorithmic choice could be incorporated in ApproxHPVM and ApproxTuner as another approximation mechanism. For example, different algorithms for specific IR-level operations could be mapped to different accuracy levels in the backend. OpenTuner [78] provides a general autotuning framework for programs written in conventional programming languages, extending the search space exploration strategies used in PetaBricks.

The *SiblingRivalry* [80] system uses the PetaBricks programming language to define a model for online autotuning that allows programs to continuously adapt to the changing environment. ApproxTuner supports a wide range of approximations, provides development-time, install-time and runtime (adaptive) autotuning, and ensures application portability.

Loop perforation [81, 82] uses autotuning to automatically detect which loops to approximate. The autotuning search consists of two phases: 1) sensitivity testing, which checks whether the perforated loop will cause the program to fail in an unacceptable way: crash the program, slow it down, cause memory leaks, or produce illegal outputs, and 2) accuracy tuning, which finds the loops with maximum speedup for every end-to-end accuracy loss bound. ACCEPT [31] uses a similar autotuning strategy, while also using source-level approximation annotations (developer annotations and type system from EnerJ). The annotations are used to restrict the search space, in order to determine approximation methods to apply between loop perforation, synchronization elision, and neural acceleration. Thus, ACCEPT needs programmer input to drive the approximation choices.

These systems do not decouple hardware-independent from hardware-dependent tuning and only consider the end-to-end quality metric. ApproxHPVM includes hardware agnostic accuracy metrics at the IR level, which capture the allowable difference between the exact and the approximate computation, which enables ApproxHPVM to achieve portable object code for approximate hardware. ApproxTuner employs a three step approximation tuning strategy, enabling hardware independent, development time software approximation options to be shipped as part of the virtual ISA, hardware-specific tuning to occur at install time, and dynamic approximation tuning at runtime.

Approximate Hardware Accelerators and Hardware Approximation Techniques. Recently there have been many proposals for machine learning accelerators. [83, 84, 85, 86]. These accelerators exploit the commonly-used computational and communication patterns

in ML applications. The DianNao project proposed small footprint accelerators that provided high-throughput machine learning computations. Recently, Google developed Tensor Processing Unit (TPU) [4], a special-purpose ASIC for accelerating neural network inference, offering high throughput for matrix operations compared to GPUs. Esmaeilzadeh et al [87, 88] proposed the Neural Processing Unit (NPU) that uses analog computation circuitry to accelerate neural network computations. Their work showed that general purpose applications can be algorithmically transformed into a neural representation, thereby facilitating execution on an NPU. Overall, these accelerators provide the same high-level characteristics as PROMISE: they accept some set of topologies and approximate operations, and efficiently compute the output. We demonstrate the support for most common matrix operations and metrics in the ApproxHPVM framework. Although we have chosen PROMISE as an approximate accelerator that exposes hardware approximation options, our framework is more broadly applicable to the wide range of emerging approximate accelerator platforms that provide hardware approximation techniques.

Precision tuning is also utilized, as it can greatly impact the achieved performance depending on the target hardware, e.g., fixed point quantization of pretrained deep neural networks using analytical methods [89] or layer-grained analytical models for bit-widths of weights and activations [90], tuning of floating-point operations [91, 92].

Software Approximations. Many studies have introduced novel software techniques for approximation that reduce execution time and/or energy. The transformations include task skipping [93, 94, 95], loop perforation [81, 82, 96, 40], approximate function substitution [38, 42, 97, 98], dynamic knobs [35] (dynamically changing function version), reduction sampling [97, 98], input data sampling [99], tuning floating-point operations [91, 92], selective discarding of atomics [37] and approximate parallelization [100, 101, 98, 102].

These techniques have been shown effective across a variety of application domains resilient to small errors. Additionally, domain specific techniques ([103]) have been proposed.

ApproxTuner, as a general framework for adaptive approximations, is highly extensible to incorporate both these software and hardware-specific approximation techniques.

Approximation Driven Adaptive Systems. *PowerDial* [35] dynamically adapts the behavior of applications in response to power or load fluctuations. It requires the application to expose the parameters that control the approximation level and uses them as knobs to control the dynamic behavior. PowerDial is limited to applications structured as an initialization phase followed by a main control loop. *JouleGuard* [36] is a runtime control system that provides guarantees of energy consumption, by dynamically configuring the system and application, while maximizing accuracy. *SAGE* [37] monitors output quality deviations at run-time, invoking more precise versions of the computation when quality is

deemed too low. *Green* [38] provides statistical Quality of Service (QoS) guarantees, utilizing loop and function approximation to generate approximate versions of expensive loops and functions, and monitors the observed QoS to ensure that the QoS requirement is met.

SpeedPress [39] uses loop perforation in the context of dynamic adaptation, to generate code that can dynamically change between different variants as the execution continues. *Sculptor* [40] uses one approximation technique, selective dynamic loop perforation, to automatically transform loops to skip selected instructions in selected iterations. These systems utilize one or a limited number of approximations, they do not separate out development-time and install-time tuning opportunities, and they do not target heterogeneous hardware with diverse approximation opportunities.

MCDNN [104] is an optimizing runtime system that uses a catalog of trained models with varying speedup and accuracy trade-offs, and switches the models at runtime based on performance/accuracy constraints. *ApproxNet* [105] introduces the spatial pyramid pooling layer (SPP) that allows for skipping certain convolution layers in the same DNN model. MCDNN is impractical for edge systems since it requires having an ensemble of large DNN models in memory while ApproxNets need model changes and hence an expensive retraining step.

CHAPTER 3: THE HPVM REPRESENTATION

Aiming to alleviate the key programmability challenges of heterogeneous systems, we propose a parallel program representation designed to provide performance portability for wide range or popular parallel hardware, including multicore CPUs, GPUs, vector hardware and potentially FPGAs.

The representation, which we call Heterogeneous Parallel Virtual Machine (HPVM) [106], is based on a hierarchical dataflow graph with side-effects. This parallelism model was selected as the basis of HPVM after carefully evaluating how it enables efficient mapping down to hardware features commonly present within hardware within that range.

This chapter describes the Heterogeneous Parallel Virtual Machine parallel program representation. The next chapter describes a specific realization of Heterogeneous Parallel Virtual Machine on top of the LLVM compiler IR.

Figures 3.1 and 3.2 show the HPVM version of a Laplacian estimate computation of a greyscale image, used as part of image processing filters. The Laplacian is a measure of the magnitude of changes in the image's brightness, and thus is used as part of image processing filters. The estimate is computed by applying a dilation filter and an erosion filter in the input image and then computing a linear combination of the initial, the dilated and the eroded image. This will be used as a running example.

3.1 HPVM PROGRAM

An HPVM program is a combination of host code plus a set of one or more distinct dataflow graphs. Each dataflow graph (DFG) is a hierarchical graph with side effects. The DFG must be acyclic. Nodes represent units of execution, and edges between nodes describe the explicit data transfer requirements. A node can begin execution once a *data item* becomes available on every one of its input edges. Repeated transfer of data items between nodes (if more inputs are provided) yields a pipelined execution of different nodes in the graph. The execution of a DFG is initiated and terminated by host code that launches the graph. For example, this mechanism can be used for streaming computations on data streams, e.g., processing successive frames in a video stream. Nodes may access globally shared memory through load and store instructions (side-effects), since hardware shared memory is increasingly common across heterogeneous systems. Because of these side effects, HPVM is not a "pure dataflow" model.

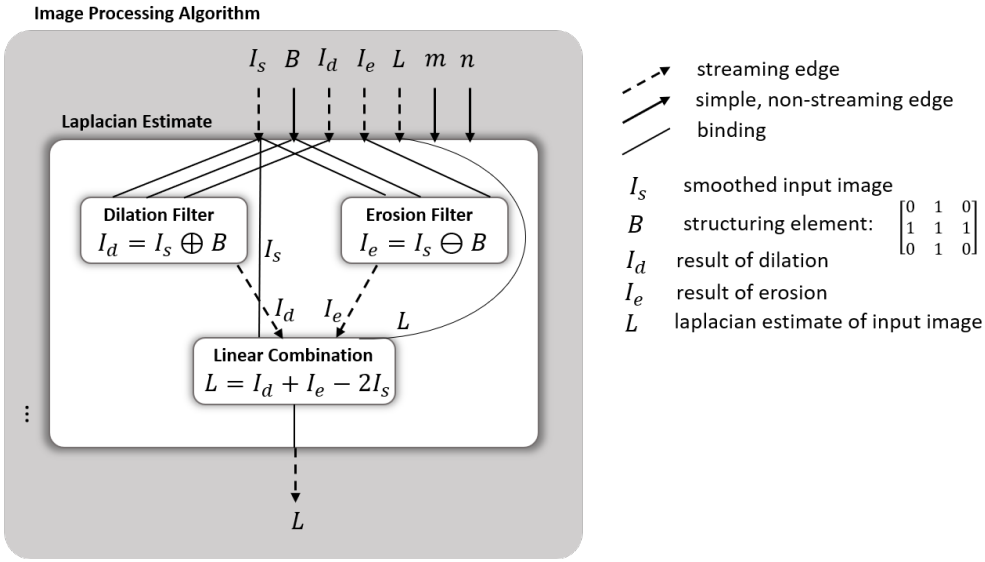


Figure 3.1: HPVM dataflow graph for a non-linear Laplacian computation

3.2 DATAFLOW NODE

A *dataflow node* represents unit of computation in the DFG. A node can begin execution once a data item becomes available on every one of its input edges. Figure 3.1 shows the components of the Laplacian as separate dataflow nodes – `Dilation Filter (DF)`, `Erosion Filter (EF)` and `Linear Combination (LC)` – connected by edges. Additionally, figure 3.2 shows the code for node `LC` and partial code for node `LaplacianEstimate`, which is standard LLVM IR except for the new intrinsic functions named `llvm.hpvm.*`. These are used to implement the HPVM abstraction in our prototype system, and are explained in subsequent sections 4.1. Load/store instructions access shared memory, using pointers that must be received explicitly from preceding nodes.

3.2.1 Dynamic Instances of Nodes

A DFG in HPVM can describe varying (data-dependent) degrees of parallelism at each node. In particular, a single static dataflow node represents multiple dynamic instances of the node, each executing the same code with different index values used to uniquely identify each dynamic instance w.r.t. the others. The dynamic instances of a node may be executed concurrently, and any required synchronization must imposed using *HPVM synchronization operations*. The dynamic instances form an n-dimensional grid, with integer indexes in each dimension, accessible via HPVM operations described below.

For example, the `LC` node in the example is replicated to have $dim_X \times dim_Y$ instances,

```
%struct.1 = type {float*}
define %struct.1 @lincomb(float* %Is, float* %Id, float* %Ie, float* %L) {
  %N = call i8* @llvm.hpvm.getNode()
  %idx = call i64 @llvm.hpvm.getNodeInstanceID.x(i8* %N)
  %idy = call i64 @llvm.hpvm.getNodeInstanceID.y(i8* %N)
  ; Index and base address calculation using %idx, %idy ...
  ; similar for Is, Id, Ie, L ...
  %pixel_Is = load float* %Is_base

  %mul = mul float 2.0, %pixel_Is
  %add = add float %pixel_Id, %pixel_Ie
  %res = sub float %add, %mul
  store float %res, float* L_base

  %out = insertvalue %struct.1 undef, float* %L, 0
  ret %struct.1 %out
}

%struct.2 = type {float*}
define %struct.2 @laplacianEstimate(float* Is, float* B, float* Id,
                                    float* Ie, float* L, i32 %m, i32 %n) {

  %dilationNode = call i8* @llvm.hpvm.createNode2D(%dilationNode, %m, %n)
  %erosionNode = call i8* @llvm.hpvm.createNode2D(%erosionNode, %m, %n)
  %lincombNode = call i8* @llvm.hpvm.createNode2D(%lincomb, %m, %n)

  ; Binding the parent input to inputs of the leaf nodes
  ;  ... input binds for dilation and erosion nodes
  call void @llvm.hpvm.bind.input(i8* %lincombNode, 0, 0, 1) ; Is
  call void @llvm.hpvm.bind.input(i8* %lincombNode, 4, 3, 1) ; L

  ; Creating dataflow edges between different nodes within same parent node
  call void @llvm.hpvm.createEdge(i8* %dilationNode, i8* %lincombNode, 1, 0, 1, 1) ; Id
  call void @llvm.hpvm.createEdge(i8* %erosionNode, i8* %lincombNode, 1, 0, 2, 1)  ; Ie

  ; Binding node outputs to parent node output
  call void @llvm.hpvm.bind.output(i8* %lincombNode, 0, 0, 1)
}
```

Figure 3.2: HPVM IR for parts of the non-linear Laplacian computation in figure 3.1

where dim_X and dim_Y are computed at runtime.

3.2.2 Dataflow Node Hierarchy

Each dataflow node in a DFG can either be a *leaf node* or an *internal node*. An internal node contains a complete DFG, called a *child graph*, and the child graph itself can have internal nodes and/or leaf nodes.

In Figure 3.1, the node `Laplacian Estimate` is an internal node, and its child graph comprises the leaf nodes `DF`, `EF`, and `LC`.

Leaf nodes contain code expressing actual computations. Leaf nodes may contain *HPVM query operations* to query the structure of the underlying DFG, and any non host side *HPVM*

operation for synchronization and memory allocation.

Internal nodes only describe the structure of the child graph, and cannot include actual computation. The internal nodes are traversed by the translators to construct a static graph and generate code for the leaf nodes and edges (section 4.2).

One restriction of this model is that the dataflow graph cannot be modified at runtime, e.g., by data-dependent code, dynamically spawning new nodes; this enables fully-static optimization and code generation at the cost of some expressivity.

Benefits of Graph Hierarchy: The grouping and hierarchy of parallelism has several advantages. It helps express several different kinds of parallelism in a compact and intuitive manner: coarse-grain task (i.e., pipelined) parallelism via top-level nodes connected using dataflow edges; independent coarse- or fine-grained data parallelism via dynamic instances of a single static node; and fine-grained data parallelism via vector instructions within single instances of leaf nodes. It provides a flexible and powerful mechanism to express tiling of computations for memory hierarchy in a portable manner (section 4.4.3). It enables efficient scheduling of the execution of the dataflow graph by grouping together appropriate sets of dataflow nodes. For example, a runtime scheduler could choose to map a single top-level (internal) node to a GPU or to one core of a multicore CPU, instead of having to manage potentially large numbers of finer-grain nodes. Finally, it supports a high-degree of modularity by allowing separate compilation of parallel components, represented as individual dataflow graphs that can be composed into larger programs.

3.3 DATAFLOW EDGE

A *dataflow edge* from the output *out* of a source dataflow node `Src` to the input *in* of a sink dataflow node `Dst` describes the explicit data transfer requirements. `Src` and `Dst` nodes must belong to the same child graph, i.e. must be children of the same internal node.

An edge from source to sink has the semantics of copying the specified data from the source to the sink after the source node has completed execution. Depending on where the source and sink nodes are mapped, the dataflow edge may be translated down to an explicit copy between compute units, or communication through shared memory, or simply a local pointer-passing operation.

The pairs `(Src,` *out*`)` and `(Dst,` *in*`)`, representing source and sink respectively, must be unique w.r.t. every other edge in the same child graph, i.e. two dataflow edges in the same child graph cannot have the same source or destination.

Hardware feature	Typical HPVM representation
GPUs	
GPU Threads	DFG leaf nodes
GPU Thread Blocks	Parent nodes of DFG leaf nodes representing GPU threads / skipped
Grid of Thread Blocks (SMs)	Parent node of DFG internal nodes representing GPU Thread Blocks / parent node of DFG leaf nodes representing GPU Threads
GPU registers, private memory	Virtual registers in LLVM code for leaf nodes
GPU Scratchpad Memory	Memory allocated in DFG internal nodes representing thread blocks
GPU Global and Constant Memory	Other memory accessed via loads and stores in DFG leaf nodes
Short-vector SIMD instructions	
Vector instructions with independent operations	Groups of dynamic instances of leaf nodes
Vector registers	Virtual registers in LLVM code for leaf nodes
Homogeneous host multiprocessor	
CPU threads in a shared-memory multiprocessor	One or more top level nodes in one or more DFGs
Shared memory	Memory accessed via loads and stores in DFG leaf nodes. HPVM operations for synchronization.
Heterogeneous multiprocessor system	
Major hardware compute units, e.g., CPU cores, GPUs	Top-level nodes in the DFG and edges between them

Table 3.1: How HPVM represents major parallel hardware features

3.3.1 Dynamic Instances of Edges

As with dataflow nodes, a static edge also represents multiple dynamic instances of that edge between the dynamic instances of the source and the sink nodes. An edge can be instantiated at runtime using one of two simple replication mechanisms:

- "all-to-all", where all dynamic instances of the source node are connected with all dynamic instances of the sink node, thus expressing a synchronization barrier between the two groups of nodes, or
- "one-to-one", where each dynamic instance of the source dataflow node is connected with a single corresponding instance of the sink node. One-to-one replication requires that the grid structure (number of dimensions and the extents in each dimension) of the source and sink nodes be identical.

Figure 3.1 shows the dataflow edges describing the data movement of smoothed input image I_s, dilated image I_d, eroded image I_e, structuring element B and output Laplacian Estimate between dataflow nodes.

Some edges (e.g., input B to node `Laplacian Estimate`) are "*fixed*", i.e. non-streaming, edges: their semantics is as if they repeatedly transfer the same data for each node execution. In practice, they are treated as a constant across node executions, which avoids unnecessary data transfers (after the first execution on a device) when executing a streaming DFG.

3.4 INPUT AND OUTPUT BIND

An internal node is responsible for mapping its inputs, provided by incoming dataflow edges, to the inputs to one or more nodes of the child graph. An internal node binds its input *ip* to input *ic* of its child node `Dst` using an input bind. The pair `(Dst,` *ic*`)` must be unique, i.e. no two input binds in the same graph can have the same destination, as that would create a conflict. Semantically, these represent name bindings of input values and not data movement.

Conversely, an internal node binds output *oc* of its child node `Src` to its output *op* using an output bind. The pair `(Src,` *oc*`)` and destination *op* must be unique, i.e. no two output binds in the same graph can have the same source destination, as that would create a conflict. A bind is always *all-to-all*.

In figure 3.1, we show the binds as undirected edges.

3.5 INTEGRATION WITH HOST CODE

In an HPVM program, the host code is responsible for setting up, initiating the execution and blocking for completion of a DFG. The host can interact with the DFG to sustain a streaming computation by sending all data required for, and receiving all data produced by, one execution of the DFG. The list of actions that can be performed by the host is described below:

- **Initialization and Cleanup:** All HPVM operations must be enclosed by the HPVM initialization and cleanup. These operations perform initialization and cleanup of runtime constructs that provide the runtime support for HPVM.
- **Track Memory:** Memory objects that are passed to dataflow graphs need to be managed by the HPVM runtime. The dataflow graph semantics of the virtual ISA assumes a globally addressable memory model. However, many accelerators present in an SoC do not support this model, e.g. discrete GPUs cannot address CPU memory directly. In such a scenario, all data has to be explicitly transferred to the target

memory space before one initiates computation on the respective compute unit. The HPVM runtime includes a "memory tracker" for tracking the location of HPVM-managed memory objects between these address spaces. Track memory inserts the specified memory object in the memory tracker and starts tracking it.

- **Untrack Memory:** Stop tracking specified memory object and remove it from memory tracker.
- **Request Memory:** If the specified memory object is not present in host memory, copy it to host memory.
- **Launch:** The host code initiates execution of specified DFG, either streaming or non streaming. This is a non-blocking operation.
  - Non streaming DFG: The host provides all data items required for execution of the DFG at the time of the launch.
  - Streaming DFG: No data is provided by the launch operation. Streaming execution is sustained by push and pop operations, described below.
- **Push:** Push a set of data items required for one graph execution to the specified DFG. The DFG must have been launched using a streaming launch operation. This is a blocking operation.
- **Pop:** Read data produced from one execution of the specified DFG. The DFG must have been launched using a streaming launch operation. This is a blocking operation.
- **Wait:** The host code blocks for completion of specified DFG.
  - For a non-streaming DFG, the data produced by the DFG are ready to be read by the host.
  - For a streaming DFG, no more data may be provided for processing by the DFG.

3.6 HPVM REPRESENTATION OF MAJOR PARALLEL HARDWARE FEATURES

An important consideration in the design of HPVM is to enable efficient mapping of code to key features of various target hardware. We focus on three kinds of parallel hardware in this work: GPUs, vectors, and multithreaded CPUs. Table 3.1 describes how the key features of these three hardware families are captured using HPVM constructs. Mapping

from HPVM constructs to the hardware features is the role of the translators described in section 4.2.

The table is a fairly comprehensive list of the major hardware features used by parallel computations, showing that HPVM is effective at capturing different kinds of hardware. We briefly describe the HPVM representation of each hardware feature.

- GPU threads: GPU threads are launched in (usually) massive numbers to execute a kernel function. In HPVM, the computation of the kernel function is represented by a leaf dataflow node. The dynamic instantiation mechanism and graph hierarchy are utilized to issue the appropriate number of GPU threads.

- GPU Thread Blocks: GPU threads within a thread block are typically allowed to perform thread block-wide synchronization and cooperate through scratchpad memory, while such operations are not permitted for all threads within a GPU kernel.

  If a kernel function utilizes such features, we opt to explicitly represent the thread blocks in the HPVM representation of the program using an internal node, and utilize the graph hierarchy to represent the intended grouping of threads. The dynamic instances of the thread block node are the number of threads, or the dimensions of the thread block, in the launched GPU kernel.

- Grid of Thread Blocks (SMs): The grid of thread blocks simply specifies how many thread blocks should be launched for a particular kernel configuration. We represent this in HPVM with a dataflow node that is the parent of the thread block node, and whose dynamic instances specify the number of thread blocks.

  If a kernel function does not utilize such features that make use of the isolation and cooperation between threads belonging in different thread blocks, then we opt for a simpler graph hierarchy that does not explicitly represent the thread blocks with separate dataflow nodes. Instead, the grid of thread blocks node is the parent of the thread node, and its dynamic instances specify the total number of threads.

- GPU registers, private memory: The thread node contains LLVM code for the kernel function. Virtual registers in the kernel function represent memory accessible only to each thread.

- GPU Scratchpad memory: As described, a kernel may utilize scratchpad memory for communication and cooperation within threads of a thread block. To represent the fact that it is only visible to the threads of a particular thread block, we allocate

memory within the dataflow node that represents the thread block, and a dataflow edge transfers it to the leaf node representing the threads. Therefore, the allocated memory from a dynamic instance of the thread block internal node for a single dynamic instance (representing one thread block) is only visible to the group of threads that are created as its own dynamic instances, and thus is private to them, achieving the desired effect.

- GPU Global and Constant memory: Global and constant memory is visible to all threads of a GPU kernel. It is provided to the grid, thread block and thread nodes through binds and/or edges, and is accessed though load and store instructions in the kernel function that is part of the thread (leaf) node.

- Vector instructions with independent operations: (Short) Vector instructions perform the same operation on all the data across the vector lanes. In HPVM, we represent a short vector instruction with no dependencies across vector lanes by grouping the LLVM instructions across as many dynamic instances of a leaf node as the vector length into a single vector instruction.

- Vector registers: Hardware vector registers are represented by the register operands and results of the vector instructions in the LLVM code of the leaf nodes.

- CPU threads in a shared memory multiprocessor: CPU threads in a shared memory multiprocessor can execute concurrently and exploit task parallelism in applications. In HPVM, we represent them using different top level nodes, that may belong in one or more DFGs. The hardware threads are utilized to execute the functionality of the one or more nodes that are mapped to them, exploiting task and/or pipeline parallelism available in the HPVM representation.

- Shared memory: In shared memory systems explicit memory transfers are not required. Dataflow edges provide DFGs with pointers to shared memory. Memory accessible through these pointers is accessed through load and store instructions in DFG leaf nodes. Synchronization is handled through appropriate HPVM operations.

A heterogeneous multiprocessor system may contain major compute units of the listed types, e.g. multicore CPUs and GPUs, in different combinations. HPVM represents all such units and their corresponding features as top level nodes in a DFG as described. A compute unit can be used to execute one or more nodes, in one or more DFGs. Dataflow edges between these nodes represent explicit data transfer requirements between the compute

units. Depending on the mapping between nodes and compute units, the dataflow edges may denote different data transfer mechanisms (refer to section 3.3).

CHAPTER 4: THE HPVM SYSTEM

HPVM supports three important capabilities that enhance the programmability of heterogeneous systems: a compiler intermediate representation (IR), a virtual instruction set (ISA), and a basis for runtime scheduling. As a compiler IR, HPVM enables effective code generation and optimization for heterogeneous systems. As a virtual ISA, it can be used to ship executable programs, in order to achieve both functional portability and performance portability across such systems. At runtime, HPVM enables flexible scheduling policies, both through the graph structure and the ability to map individual nodes in a program to any of the target devices on a system. As proof of concept, we have implemented a prototype HPVM system, which we describe in this chapter.

4.1 HPVM VIRTUAL ISA AND COMPILER IR

Based on the HPVM abstraction as defined in chapter 3, we have developed a specific realization of HPVM on top of LLVM, also called HPVM. Our prototype system includes a compiler IR, a virtual ISA, an optimizing compiler, and a runtime scheduler, all based on the HPVM representation as described in chapter 3.

The compiler IR is an extension of the LLVM IR, defined via LLVM intrinsic functions, and supports both code generation (section 4.2) and optimization (section 4.4) for heterogeneous parallel systems. The virtual ISA is essentially just an external, fully executable, assembly language representation of the compiler IR.

We define new instructions for all the HPVM abstractions as specified in chapter 3, describing and querying the structure of the dataflow graph, for memory management and synchronization, as well as for host operations initiating, terminating and interacting with the execution of a graph. We express the new instructions as function calls to newly defined LLVM "*intrinsic functions*", a standard LLVM mechanism to extend the instruction set and communicate information to a particular back end. A call to an intrinsic function appears to existing LLVM passes as a function call to an unknown external function. This ensures that existing passes do not need to be modified to correctly compile code containing calls to new intrinsics.

The intrinsic functions used to define the HPVM compiler IR and virtual ISA are shown in section 4.1.1. The code for each dataflow node is given as a separate LLVM function called the "*node function*," specified as function pointer `F` for applicable intrinsics `llvm.hpvm.createNode[1D,2D,3D]`, `llvm.hpvm.launch`. The node function may call ad-

ditional, "auxiliary" functions. However, the auxiliary functions are not allowed to include any HPVM intrinsics, as they are not considered to be part of the HPVM node hierarchy. The incoming dataflow edges and their data types are denoted by the parameters to the node function. The outgoing dataflow edges are represented by the return type of the node function, which must be an LLVM struct type with zero or more fields (one per outgoing edge).

Pointer arguments of node functions are required to be annotated with attributes `in`, and/or `out`, depending on their expected use (read only, write only, read write), as explained further in section 4.2.

Each top-level dataflow graph (DFG) in an HPVM program is defined by its own root node function which creates the underlying DFG structure. The DFG is the (internal) root node's child graph. Unlike regular internal nodes, the root node only has one dynamic instance because it instantiates the top-level DFG. The DFG is launched by the host code using the root node function, as described in section 4.1.1.

In order to manipulate or query information about graph nodes and edges, we represent

- nodes with opaque handles (pointers of LLVM type i8*).
- inputs of a node as integer indices into the list of function arguments of the node function. Figure 3.2 shows that node `LC` has four input arguments, I_s, I_d, I_e and L in this order. We use the integers 0, 1, 2 and 3 respectively to transfer data to them through dataflow edges or bind the parent node's data.
- outputs of a node as integer indices into the return struct type of the node function. Node `LC` has one output, L, that has index 0 in the return struct type. We will use this index when an edge or a bind must use the output of `LC`.

A few additional insights and observations for the intrinsic functions are provided here. Firstly, using LLVM functions for node code makes HPVM an "outlined" representation, and the function calls interfere with existing intraprocedural optimizations at node boundaries.However, offloading computation to accelerators raises different tradeoffs for optimizations across parallelism constructs, e.g. data transfer vs redundant computation, and care should be taken before applying intraprocedural optimizations designed for sequential programs. Secondly, note that although the object returned from `llvm.hpvm.malloc` can be shared by all nodes, it must somehow be communicated explicitly for use by other nodes. Finally, `llvm.hpvm.barrier` only synchronizes the dynamic instances of the node that executes it, and not all other concurrent nodes. In particular, there is no global barrier operation

in HPVM, but the same effect can be achieved by merging dataflow edges from all concurrent nodes.

4.1.1 HPVM Intrinsics

This section defines the intrinsic functions used to implement the HPVM internal representation. The notation `i`N represents the N-bit integer type in LLVM.

Intrinsics for Describing Graphs The intrinsics for describing graphs can only be "executed" by internal nodes; *all these intrinsics are interpreted by the compiler at code generation time and erased*, effectively fixing the graph structure. (Only the number of dynamic instances of a node can be varied at runtime.) Additionally, internal nodes are only allowed to have intrinsics for describing graphs as part of their node function, with the exception of a return statement of the appropriate type, in order to return the result of the outgoing dataflow edges.

Conversely, all other intrinsics are executable at run-time, and can only be used by leaf nodes or by host code.

`i8* llvm.hpvm.createNode(i8* F)` Create a static dataflow node with one dynamic instance executing node function `F`. Return a handle to the created node.

`i8* llvm.hpvm.createNode1D(i8* F, i64 n1)` Create a static dataflow node replicated in one dimension, namely `x`, with n_1 dynamic instances executing node function `F`. Return a handle to the created node.

`i8* llvm.hpvm.createNode2D(i8* F, i64 n1, i64 n2)` Create a static dataflow node replicated in one dimension, namely `x` and `y`, with n_1 and n_2 dynamic instances in each dimension respectively, executing node function `F`. Return a handle to the created node.

Figure 3.2 shows the usage of this intrinsic to create the nodes `DF`, `EF` and `LC` as part of the node function of their parent node.

`i8* llvm.hpvm.createNode3D(i8* F, i64 n1, i64 n2, i64 n3)` Create a static dataflow node replicated in one dimension, namely `x`, `y` and `z`, with n_1, n_2 and n_3 dynamic instances in each dimension respectively, executing node function `F`. Return a handle to the created node.

`i8* llvm.hpvm.createEdge(i8* Src, i8* Dst, i1 ReplType, i32 sp, i32 dp, i1 isStream)` Create edge from output *sp* of node `Src` to input *dp* of node `Dst`. Argument *dp* of `Dst`'s node function and field *sp* of the return struct in `Src`'s node function must have matching types. `ReplType` chooses between a one-to-one (0) or all-to-all (1) edge. `isStream` chooses a streaming (1) or non streaming (0) edge. Return a handle to the created edge.

Figure 3.2 shows the usage of this intrinsic to create two streaming dataflow edges between nodes `DF-LC`, transferring I_d, and `EF-LC`, transferring I_e. To illustrate the meaning of indices *sp* and *dp*, note that both edges are transferring the single output of their respective nodes. We refer to the output of a node using the index into the return struct, which in this case is 0, thus in both edges, $sp = 0$. For the destination position, I_d is at index 1 in the argument list of the node function of `LC`, and I_e at index 2. Therefore, $dp = 1$ for the first edge and $dp = 2$ for the second.

`void llvm.hpvm.bind.input(i8* N, i32 ip, i32 ic, i1 isStream)` Bind input *ip* of current node to input *ic* of child node `N`. Argument *ic* of `N`'s node function and argument *ip* of the current node function must have matching types. `isStream` chooses a streaming (1) or non streaming (0) bind.

Figure 3.2 shows the intrinsics for the input binds for node `LC`. To illustrate the meaning of indices *ip* and *ic*, I_s is at position 0 in the argument list of the node functions for both `Laplacian Estimate` and `LC`, thus in the bind for I_s $ip = 0$ and $ic = 0$. However, L is at position 4 in the argument list of the node function of `Laplacian Estimate`, and position 3 in that of `LC`, thus in the bind for L, $ip = 4$ and $ic = 3$.

`void llvm.hpvm.bind.output(i8* N, i32 oc, i32 op, i1 isStream)` Bind output *oc* of child node `N` to output *op* of current node. Field *oc* of the return struct in `N`'s node function and field *op* of the return struct in the current node function must have matching types. `isStream` chooses a streaming (1) or non streaming (0) bind.

Figure 3.2 shows the output bind used to bind the output of `LC` to the output of the `Laplacian Estimate`. The index of the output in both return structs is 0, hence *oc* and *op* are assigned to 0.

Intrinsics for Querying Graphs The following intrinsics are used to query the structure of the dataflow graph. They can only be used by leaf nodes.

`i8* llvm.hpvm.getNode()` Return a handle to the current leaf node.

`i8* llvm.hpvm.getParentNode(i8* N)` Return a handle to the parent in the hierarchy of node `N`.

`i32 llvm.hpvm.getNumDims(i8* N)` Get the number of dimensions of node `N`.

`i64 llvm.hpvm.getNodeInstanceID.[xyz](i8* N)` Get index of current dynamic node instance of node `N` in dimension `x`, `y` or `z` respectively. The dimension must be one of the dimensions in which the node is replicated.

`i64 llvm.hpvm.getNumNodeInstances.[xyz](i8* N)` Get number of dynamic instances of node `N` in dimension `x`, `y` or `z` respectively. The dimension must be one of the dimensions in which the node is replicated.

Intrinsics for Memory Allocation and Synchronization The following intrinsics are used for memory allocation and synchronization. They can only be used by leaf nodes.

`i8* llvm.hpvm.malloc(i64 nBytes)` Allocate a block of memory of size `nBytes` and return pointer to it.

`i32 llvm.hpvm.atomic.add(i8* m, i32 v)` Atomically compute the bitwise ADD of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.sub(i8* m, i32 v)` Atomically compute the bitwise SUB of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.min(i8* m, i32 v)` Atomically compute the bitwise MIN of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.max(i8* m, i32 v)` Atomically compute the bitwise MAX of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.xchg(i8* m, i32 v)` Atomically compute the bitwise XCHG of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.and(i8* m, i32 v)` Atomically compute the bitwise AND of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.or(i8* m, i32 v)` Atomically compute the bitwise OR of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`i32 llvm.hpvm.atomic.xor(i8* m, i32 v)` Atomically compute the bitwise XOR of `v` and the value stored at memory location `[m]` w.r.t. the dynamic instances of the current leaf node and stores the result back into `[m]`. Return the value previously stored at `[m]`.

`void llvm.hpvm.barrier()` *Local* synchronization barrier across dynamic instances of current leaf node, i.e., only synchronizes the dynamic instances of the node that executes it, and not all other concurrent nodes.

Intrinsics for Graph Interaction The following intrinsics are for graph initialization/termination and interaction with the host code, and can be used only by the host code.

`void llvm.hpvm.init()` Initialization of HPVM runtime.

`void llvm.hpvm.cleanup()` Cleanup of HPVM runtime created objects.

`void llvm.hpvm.trackMemory(i8* ptr, i64 sz)` Insert memory starting at `ptr` of size sz in the memory tracker. `ptr` becomes the key for identifying this memory object. As soon as a memory object is inserted in the memory tracker it starts being tracked, and can be passed as a data item to a dataflow graph.

`void llvm.hpvm.untrackMemory(i8* ptr)` Stop tracking memory object with key `ptr`, and remove it from memory tracker.

`void llvm.hpvm.requestMemory(i8* ptr, i64 sz)` If memory object with key `ptr` is not located in host memory, copy it to host memory.

`i8* llvm.hpvm.launch(i8* RootGraph, i8* Args, i1 isStream)` Launch the execution of a top-level dataflow graph with root node function `RootGraph`. The execution of the launched graph is asynchronous. `Args` is a pointer to a packed struct, containing one field per argument of the `RootGraph` function, consecutively. For non-streaming graphs with a non empty result type, `Args` must contain an additional field of the type `RootGraph.returnTy`, where the result of the graph will be returned. `isStream` chooses between a non streaming (0) or streaming (1) graph execution. Return a handle to the invoked dataflow graph.

`void llvm.hpvm.wait(i8* GraphID)` Wait for completion of execution of dataflow graph with handle `GraphID`.

`void llvm.hpvm.push(i8* GraphID, i8* args)` Push set of input data `args` (same as type described in launch) to streaming graph with handle `GraphID`.

`i8* llvm.hpvm.pop(i8* GraphID)` Pop and return data from streaming graph with handle `GraphID`. The return type is a struct containing a field for every output of the graph.

4.2 COMPILATION STRATEGY

We describe the key aspects of the compilation strategy.

4.2.1 Background

We begin with some background on how code generation works for a virtual instruction set, shown for HPVM in Figure 4.1. At the developer site, front-ends for one or more source languages lower source code into the HPVM IR. One or more optimizations may be optionally applied on this IR, to improve program performance, while retaining the IR structure and semantics. The possibly optimized code is written out in an object code or assembly language format, using the IR as a virtual ISA, and shipped to the user site (or associated server). A key property of HPVM (like LLVM [107]) is that the compiler IR and the virtual ISA are essentially identical. Once the target hardware becomes known (e.g., at the user site or server), the compiler back-end is invoked. The back-end traverses the Virtual ISA and uses one or more target-ISA-specific code generators to lower the program to executable native code.

Hardware vendors provide high-quality backends for individual target ISAs, which we can often leverage for our system, instead of building a complete native back-end from scratch

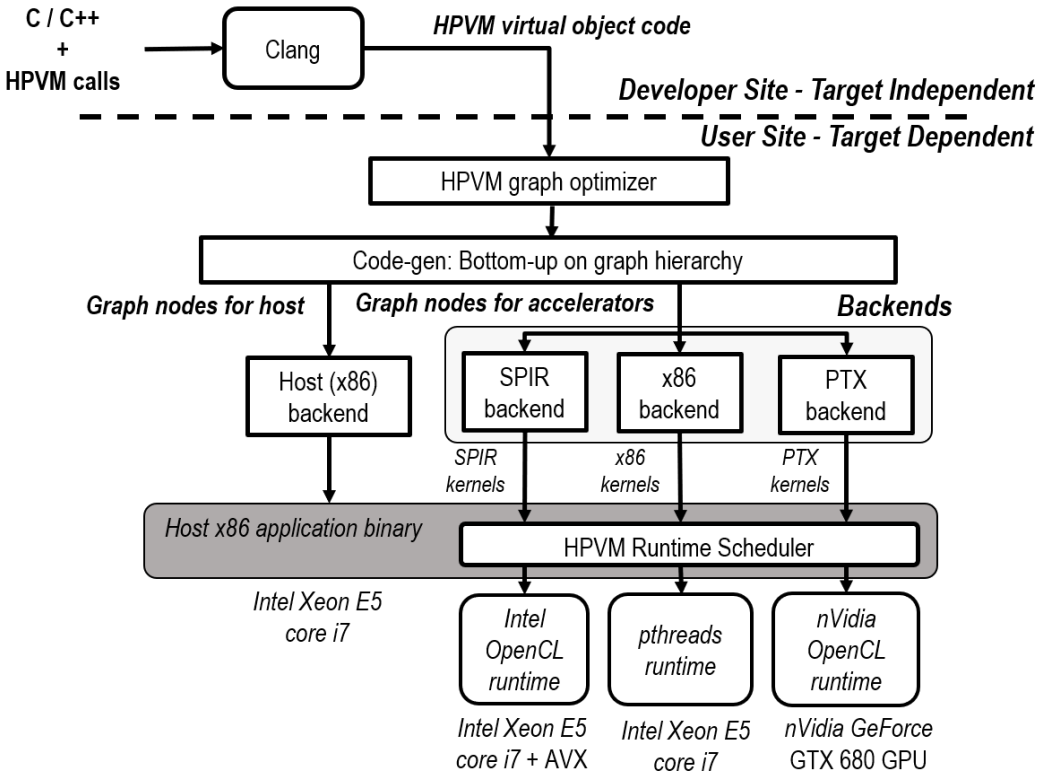


Figure 4.1: Overview of compilation flow for HPVM.

for each target. We do this for the PTX ISA on NVIDIA GPUs, AVX vector ISA for Intel processors, and X86-64 ISA for individual threads on Intel host processors.

In this work, we focus on using HPVM for efficient code generation (this section) and optimizations (section 4.4). We leave front-ends for source languages for future work. Note that we do rely on a good dataflow graph (representing parallelism, not too fine-grained nodes, good memory organization) for good code generation. This need can be met with a combination of parallelism information from suitable parallel programming languages (such as OpenMP or OpenCL), combined with the graph optimizations at the HPVM level, described in section 4.4. We do *not* rely on precise static data dependence analysis or precise knowledge of data transfers or memory accesses, which is important because it means that we can support irregular or data-dependent parallelism and access patterns effectively.

4.2.2 HPVM Compilation Flow

The HPVM compilation flow follows the structure shown in Figure 4.1.

The compiler invokes separate back-ends, one for each target ISA. Each back end performs

a depth-first traversal of the dataflow graph, maintaining the invariant that code generation is complete for all children in the graph hierarchy of a node, N, before performing code generation for N. Each back-end performs native code generation for selected nodes, and associates each translated node with a host function that implements the node's functionality on the target device.

We have implemented back-ends for three target ISAs: PTX (GPU), AVX (Vector), and X86-64 (CPU). Each backend emits a device-specific native code file that includes a device specific function per translated node. We use simple annotations on the node functions to specify the target compute unit manually, where the annotation may specify one *or more* of `GPU, Vector, CPU`. The following subsections briefly describe each backend.

4.2.3 Code Generation for PTX

The PTX [108] backend builds on the existing NVPTX back-end in LLVM. This back end translates an extended version of the LLVM IR called NVVM (containing PTX-specific intrinsic functions) [109] into PTX assembly.

A node annotated for GPU will contain a child graph that forms a one-level or two-level DFG, depending on whether or not the computation is tiled, as shown in Table 3.1 and explained in section 4.4.3, creating a combined two-level or three-level graph hierarchy. Our translator for PTX takes as input the internal node `IN` containing this DFG. It generates an NVVM kernel function for each leaf node, which will execute the dynamic instances of the leaf node. If `IN` is a three-level node, and the second (thread block) level node contains an *allocation node* (defined as a leaf node that allocates memory using the `llvm.hpvm.malloc` intrinsic), the allocated memory is assigned to scratchpad memory, as explained in section 4.4.3. All other memory is allocated by the translator to GPU global memory or GPU constant memory. We use a simple heuristic to determine when to assign a memory object to constant memory, described in section 4.4.2. The LLVM address space mechanism is used to easily perform this assignment, because the NVPTX backend interprets pointers with address space 1, 3, and 4 as pointers to global, scratchpad, and constant memory respectively.

The HPVM `llvm.hpvm.getNode()` and `llvm.hpvm.getParentNode(i8*)` intrinsics are used to find the node, in the node hierarchy of the DFG, that will be queried for information. Subsequently, the HPVM query intrinsics are translated to NVVM intrinsics. Specifically, we replace the HPVM intrinsics with OpenCL device calls that capture the meaning of the HPVM query intrinsics depending on the graph structure. An example of such mapping is that the `llvm.hpvm.getNodeInstanceID.[xyz](N)` is mapped to the OpenCL call

`get_global_id(0/1/2)` in a two level DFG with leaf node `N`, since in this case a unique identifier for each dynamic instance of `N` can be returned by the OpenCL call `get_global_id` in the respective dimension for the generated GPU kernel function. However, it is is mapped to the OpenCL call `get_local_id(0/1/2)` in a three level DFG with leaf node `N`, since in this case a unique identifier for the inner level of the hierarchy is requested, which would correspond to a thread identifier within a thread block of a GPU kernel. In general, we use the OpenCL calls `get_global_id`, `get_local_id`, and `get_group_id` to appropriately translate the HPVM query intrinsics, and the OpenCL atomics and synchronization instructions to implement the HPVM synchronization intrinsics.

Finally, we link with libclc [110], an open source implementation of the library requirements of the OpenCL C programming language on top of multiple targets including NVPTX. This step replaces the OpenCL device calls with NVVM intrinsics, and eventually generates an NVVM kernel function for the DFG. The generated NVVM kernel is translated to PTX by the NVPTX back-end.

Our translator also generates code to load and run the PTX assembly of the leaf node on the GPU, and for all data transfers for the node's and outputs. Calls are inserted to the HPVM runtime, that is implemented on top of the NVIDIA OpenCL runtime, for that purpose. Data transfer operations are only performed when needed as determined by the memory tracker (sections 4.2.6 and 4.3). This code is encapsulated in a function that becomes the host function associated with the input dataflow node `IN` on the GPU. I.e., it is a CPU function exposed to the higher level of hierarchy in the dataflow graph that can be called to implement the functionality of `IN` on the GPU device.

4.2.4 Code Generation for AVX

Dynamic instances of leaf nodes are assumed to independent and may be executed concurrently, with synchronization imposed only by HPVM intrinsics, making it possible to vectorize *across* node instances. We leverage Intel's translator from SPIR [111] (predecessor of SPIR-V. SPIR supported only OpenCL device programs, instead of compute kernels for multiple Khronos APIs) to AVX, which is part of Intel's OpenCL runtime system, for two reasons: it recognizes and utilizes the independence of SPIR work items to produce vector parallelism, and it is well tuned to produce efficient code for the AVX instruction set. Instead of writing our own AVX code-generator directly from HPVM with these sophisticated capabilities, we chose to write a translator that converts HPVM code to SPIR. The dynamic instances of leaf nodes become SPIR work items and HPVM synchronization operations are translated to SPIR. The SPIR representation is largely LLVM with OpenCL device calls and

address spaces to represent memory spaces similarly to the NVPTX backend, therefore the translation process is similar to the one described for the PTX backend, up to generating the OpenCL device calls.

The generated SPIR code is then vectorized for AVX by Intel's translator. Our translator also creates the necessary host function to initiate the execution of the SPIR kernel in a process similar to the PTX backend, generating HPVM runtime calls that invoke the Intel OpenCL runtime for execution of a SPIR kernel on a CPU device instead.

4.2.5 Host Code Generation

The x86 backend is invoked last, and is responsible for the following:

- Native code generation for all nodes annotated as CPU nodes: We build upon the LLVM X86 backend for regular LLVM IR, adding support for HPVM intrinsics. The HPVM intrinsics `llvm.hpvm.getNode()` and `llvm.hpvm.getParentNode(i8*)` are used to find the node, in the node hierarchy of the DFG, that will be queried for information. Subsequently, the HPVM query intrinsics are translated to LLVM instructions.

  Specifically, we extend the node function with 0, 2, 4 or 6 arguments depending on whether the node has 0, 1, 2 or 3 dimensions respectively, that represent a dynamic instance's unique identifier and a dimension's limit. The transformed function is the generated native function.

  We replace the HPVM intrinsics `llvm.hpvm.getNodeInstanceID.[xyz](N)` and `llvm.hpvm.getNumNodeInstances.[xyz](N)` with the corresponding argument, when its argument `N` is the leaf node. If `N` is an internal node, then these intrinsics are implemented using support from the HPVM runtime, that maintains a stack of indices and limits of nodes in the hierarchy higher than the leaf node containing the intrinsic. [1]

  For the internal nodes, we translate `createNode` operations to loops that enumerate the dynamic instances of the created node by calling the node function extended with the identifier and dimension arguments. We also insert calls to the HPVM runtime, that pushes the internal node's dynamic instance identifier and dimension limit in

[1] The HPVM memory and synchronization intrinsics are currently not supported in the x86 backend. The `llvm.hpvm.atomic.*` can be supported though LLVM atomic instructions, and `llvm.hpvm.malloc` can be implemented using malloc and be transparently cached. the `llvm.hpvm.barrier` poses a bigger challenge, as it imposes a synchronization point across all dynamic instances of a node, thereby making code generation by enumerating the dynamic instances using loops incorrect. Instead, two different loops must be generated, one for the code up to `llvm.hpvm.barrier` and one for the code after it, effectively performing loop fission.

the runtime maintained stack before the loop executing each node function, and pops them after it is completed. We translate dataflow edges to appropriate data transfers (section 4.2.6).

For top level internal nodes in streaming dataflow graphs, we additionally generate a *filter function*, that repeatedly calls the generated native function in a loop, while providing the inputs from and the outputs to the streaming edges. We generate circular buffers for each of the streaming inputs and outputs, and a new thread to execute the filter function of the node. `llvm.hpvm.push` and `llvm.hpvm.pop` intrinsics are translated to HPVM runtime calls (section 4.3).

- For nodes with multiple native versions, i.e. annotated with more than one target, generating a wrapper function that invokes the HPVM runtime scheduler (section 4.3) to choose which target function to execute, on every invocation of the node if the graph is streaming. The wrapper function has the same signature as the generated native versions. We generate code that simply calls the `getVersion` method, that encapsulates the choice of target device (section 4.3), and conditionally calls the generated native versions based on the result of this call. The wrapper function becomes the generated native function for nodes with multiple native versions.

- Host-side coordination code, enforcing the order of execution dictated by the dataflow graph. This is enforced by traversing the dataflow nodes of the child graph of each internal node in topological sort. Code generation is performed in this order, therefore satisfying the dependencies imposed by dataflow edges and respecting the dictated order of execution.

- Code to initiate and terminate execution of each dataflow graph. We launch each dataflow graph using a separate thread to execute the graph's generated root function, thus ensuring asynchronous execution of the graph and host code, and wait for the graph's execution to complete by joining with the launched thread.

4.2.6 Data Movement

Code generation for dataflow edges is performed as part of translating the internal dataflow node containing the edge. When the source and sink node execute on the same compute unit, or if they execute on two different compute units that share memory, passing a pointer between the nodes is enough. Such pointer passing is safe even with copy semantics: a

dataflow edge implies that the source node must have *completed* execution before the sink node can begin, so the data will not be overwritten once the sink begins execution.

Some accelerators including many GPUs and FPGAs, only have private address spaces and data needs to be explicitly transferred to or from the accelerator memory. In such cases, we generate explicit data copy instructions using the accelerator API, e.g., OpenCL for GPUs.

It is important to avoid unnecessary data copies between devices for good performance. To that end, we allow explicit attributes `in` and/or `out` on node arguments, and only generate the specified data movement. Achieving the same effect without annotations would require an interprocedural May-Mod analysis [112] for pointer arguments, which we aim to avoid as a requirement for such a key optimization.

4.3 HPVM RUNTIME AND SCHEDULING FRAMEWORK

Some features of our translators require runtime support.

First, the HPVM design allows a leaf node to query node instance and dimension queries to any ancestor. When such a query can be addressed by hardware registers, the query intrinsic is replaced by the corresponding accelerator API call (as shown in the PTX backend). However, when it is not supported, the runtime maintains a stack to keep track of the instance ID, and dimension limit of the dynamic instance of the ancestors and responds when a query arrives. This stack is used by the x86 backend, that utilises the stack for querying instance IDs and dimension limits of internal nodes.

Second, global memory must be shared across nodes mapped to devices with separate address spaces. To support this, the translators insert calls to the HPVM runtime, that utilize the appropriate accelerator runtime API (in our case, the OpenCL runtime) to perform the copies. Such copies are sometimes redundant, e.g., if the data has already been copied to the device by a previous node execution. The HPVM runtime includes a conceptually simple "memory tracker" to record the locations of the latest copy of data arrays, and thus avoid unnecessary copies. The memory tracker implements the track, untrack, and request memory host operations (section 3.5) for specified memory objects. Additionally, the memory tracker is implicitly invoked as part of the implementation of a dataflow edge. A request memory operation is performed from the location of the sink dataflow node, for every dataflow edge incoming to that node in order to ensure that the input data will be present at the beginning of the execution of aforementioned node. If the requested data is already present, passing a pointer is sufficient. Otherwise the appropriate accelerator runtime API calls are invoked to perform the copies.

Third, streaming edges are implemented using buffering and different threads are used to perform the computation of each pipeline stage. The required buffers, threads, and data copying are managed by the HPVM runtime. Specifically, the HPVM runtime implements a circular buffer data structure, and blocking push and pop operations on it. The threads needed for all nodes of a streaming graph that are executing the node filter functions are created when the dataflow graph is launched and destroyed when the data stream ends. All data transfers between two nodes are automatically managed. Specifically, a streaming edge is implemented as a circular buffer, and data transfers between nodes are handled by popping from and pushing to the buffers corresponding to the edges between them. The runtime calls implementing the `llvm.hpvm.push` and `llvm.hpvm.pop` intrinsics push and pop data to and from circular buffers corresponding to the inputs and outputs of the root node of the launched dataflow graph. We use an additional buffer, isLastInput, that contains a streaming boolean, to denote the end of the data stream when false.

Finally, the runtime is invoked when a runtime decision is required about where to schedule the execution of a dataflow node with multiple translations. We use a run-time policy to choose a target device, based on the dataflow node identifier, the data item number for streaming computations, and any performance information available to the runtime. (Data item numbers are counted on the host: 0 or higher in a streaming graph, -1 in a non-streaming graph.) We encapsulate the decision into a base policy class with a method `getVersion(const char *node_name, int64_t data_item_no)` that returns a target device; specific policies can be implemented by subclasses of this class. This method is invoked on a selected policy object, and the result is used to invoke the version for the selected target. This basic framework allows a wide range of scheduling policies. We have implemented a few simple static and dynamic policies:

1. *Static Node Assignment*: Always schedule a dataflow node on a fixed, manually specified target, so the target depends only on the node identifier, i.e., the first argument of `getVersion`.

2. *Static Data Item Assignment*: Schedule all nodes of a graph for a particular input data item on a single target, so the target depends only on the data item number, i.e., the second argument of `getVersion`.

3. *Dynamic*: A dynamic policy that uses the node identifier as in policy #1 above, plus instantaneous availability of each device: when a specified device is unavailable, it uses the CPU instead.

Experimenting with more sophisticated scheduling policies within the framework is out

of scope for this work. Within the scope of this work, we simply aim to show that we offer the flexibility to support flexible runtime scheduling decisions. For example, the second and third policies above could use a wide range of algorithms to select the target device per data item among all available devices. The key to the flexibility is that HPVM allows individual dataflow graph nodes to be compiled to any of the targets.

4.4 COMPILER OPTIMIZATION

An important capability of a compiler IR is to support effective compiler optimizations. The hierarchical dataflow graph abstraction enables optimizations of explicitly parallel programs at a higher (more informative) level of abstraction than a traditional IR (such as LLVM and many others), that lacks explicitly parallel abstractions; i.e., the basic HPVM intrinsics, `llvm.hpvm.createNode*`, `llvm.hpvm.createEdge`, `llvm.hpvm.getNodeInstanceID.*`, etc., are directly useful for many graph analyses and transformations. In this section, we describe a few optimizations enabled by the HPVM representation.

4.4.1 Node Fusion

One optimization we have implemented as a graph transformation is *Node Fusion*. It can lead to more effective redundancy elimination and improved temporal locality across functions, reduced kernel launch overhead on GPUs, and sometimes reduced barrier synchronization overhead. Fusing nodes, however, can hurt performance on some devices because of resource constraints or functional limitations. For example, each streaming multiprocessor (SM) in a GPU has limited scratchpad memory and registers, and fusing two nodes into one could force the use of fewer thread blocks, reducing parallelism and increasing pressure on resources.

We use a simple policy to decide when to fuse two nodes; for our experiments, we provide the node identifiers of nodes to be fused as inputs to the node fusion pass. More sophisticated node fusion policies can be developed, perhaps guided by profile information or autotuning.

Two nodes $N1$ and $N2$ are valid node fusion candidates if: (1) they both are (a) leafs, or (b) internal nodes containing an optional allocation node (refer to section 4.2.3) and a single other leaf node (which we call the *compute node*); (2) they have the same parent, target, dimensions and size in each dimension, and, if they are internal nodes, so do their compute nodes and their optional allocation nodes; and (3) they are either concurrent (no path of edges connects them), or they are connected directly by one-to-one edges and there is no data transfer between $N1$'s compute and $N2$'s allocation node, if any.

The result is a fused node with the same internal graph structure, and with all incoming (similarly, outgoing) edges of $N1$ and $N2$, except that edges connecting $N1$ and $N2$ are replaced by variable assignments.

Note that fusing nodes may reduce parallelism, or may worsen performance due to greater peak resource usage. Nodes that have been fused may need to be split again due to changes in program behavior or resource availability, but fusing nodes loses information about the two original dataflow nodes. More generally, node splitting is best performed as a first-class graph transformation, that determines what splitting choices are legal and profitable instead of keeping track of which nodes have been previously merged as split candidates. We leave this transformation to future work.

4.4.2 Mapping Data to GPU Constant Memory

GPU global memory is highly optimized (in NVIDIA GPUs) for coalescing of consecutive accesses by threads in a thread block: irregular accesses can have orders-of-magnitude lower performance. In contrast, constant memory is optimized for read-only data that is invariant across threads and is much more efficient for thread-independent data.

The HPVM translator for GPUs automatically identifies data that should be mapped to constant memory. The analysis is trivial for scalars, but also simple for array accesses because of the HPVM intrinsics: for array index calculations, we identify whether they depend on (1) the `getNodeInstanceId.*` intrinsics, which is the sole mechanism to express thread-dependent accesses, or (2) memory accesses. Those without such dependencies are uniform and are mapped to constant memory, and the rest to GPU global memory. The HPVM translator identified such candidates in 3 (spmv, tpacf, cutcp) out of 7 benchmarks, resulting in 34% performance improvement in tpacf and no effect on performance of the other two benchmarks.

4.4.3 Memory Tiling

The programmer, an optimization pass or a language front-end can "tile" the computation by introducing an additional level in the dataflow graph hierarchy. The (1D, 2D or 3D) instances of a leaf node would become a single (1D, 2D or 3D) tile of the computation. The (1D, 2D or 3D) instances of the parent node of the leaf node would become the (1D, 2D or 3D) blocks of tiles.

Memory can be allocated for each tile using the `llvm.hpvm.malloc` intrinsic in a single allocation node (refer to section 4.2.3), which passes the resulting pointer to all instances of

the leaf node representing the tile. This memory would be assigned to scratchpad memory on a GPU orget transparently cached on the CPU.

In this manner, *a single mechanism, an extra level in the hierarchical dataflow graph, represents both tiling for scratchpad memory on the GPU and tiling for cache on the CPU*, while still allowing device-specific code generators or autotuners to optimize tile sizes separately. On a GPU, the leaf node becomes a thread block and we create as many thread blocks as the dimensions of the parent node. On a CPU or AVX target, the code results in a loop nest with as many blocks as the dimensions of the parent node, of tiles as large as the dimensions of the leaf node.

We have used this mechanism to create tiled versions of four of the seven Parboil benchmarks evaluated in section 5. The tile sizes are determined by the programmer in our experiments. For the three benchmarks (`sgemm, tpacf, bfs`) for which non-tiled versions were available, the tiled versions achieved a mean speedup of 19x on GPU and 10x on AVX, with `sgemm` getting as high as 31x speedup on AVX.

CHAPTER 5: HPVM EVALUATION

We evaluate the HPVM representation by examining several questions:

1. Is HPVM performance-portable: can we use the *same virtual object code* to get "good" speedups on different compute units, and how close is the performance achieved by HPVM compared with hand-written OpenCL programs?
2. Does HPVM enable flexible scheduling of the execution of target programs?
3. Does HPVM enable effective optimizations of target programs?

5.1 EXPERIMENTAL SETUP AND BENCHMARKS

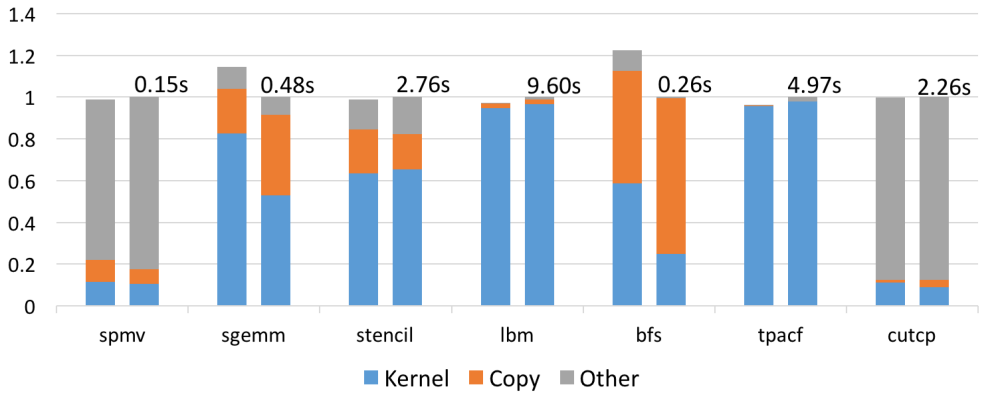
We define a set of C/C++ functions corresponding to the HPVM intrinsics and use them to write parallel HPVM applications. We implemented a simple HPVM C/C++ frontend to generate the virtual ISA from this representation.

We translated the *same* HPVM code to two different target units: the AVX instruction set in an Intel Xeon E5 core i7 and a discrete NVIDIA GeForce GTX 680 GPU card with 2GB of memory. The Intel Xeon also served as the host processor, running at 3.6 GHz, with 8 cores and 16 GB RAM.

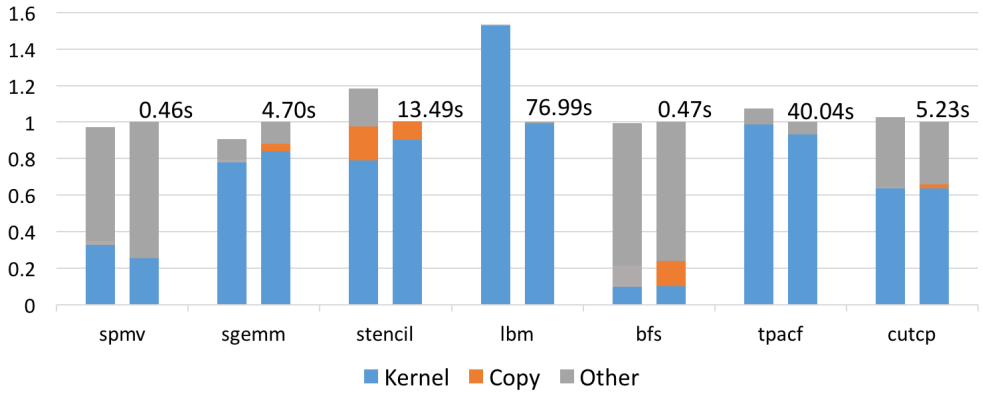
For the performance portability and hand-coded comparisons, we used 7 OpenCL applications from the Parboil benchmark suite [113]: Sparse Matrix Vector Multiplication (spmv), Single-precision Matrix Multiplication (sgemm), Stencil Partial Differential Equation solver (stencil), Lattice-Boltzmann (lbm), Breadth-first search (bfs), Two Point Angular Correlation Function (tpacf), and Distance-cutoff Coulombic Potential (cutcp).

In the GPU experiments, our baseline for comparison is the best available OpenCL implementation. For spmv, sgemm, stencil, lbm, bfs and cutcp, that is the Parboil version labeled `opencl_nvidia`, which has been hand-tuned for the Tesla NVIDIA GPUs [114]. For tpacf, the best is the generic Parboil version labeled `opencl_base`. We further optimized the codes by removing unnecessary data copies (bfs) and global fences (tpacf, cutcp). All the applications are compiled using NVIDIA's proprietary OpenCL compiler.

In the vector experiments, with the exception of stencil and bfs, our baseline is the same OpenCL implementations we chose as GPU baselines, but compiled using the Intel OpenCL compiler, because these achieved the best vector performance as well. For stencil, we used `opencl_base` instead, as it outperformed `opencl_nvidia`. For bfs, we also used `opencl_base`, as `opencl_nvidia` failed the correctness test. The HPVM versions were generated to match



(a) GPU Experiments - Normalized Execution Time.



(b) Vector Experiments - Normalized Execution Time.

Figure 5.1: For each benchmark, left bar is HPVM and right bar is OpenCL baseline. The absolute time of the baseline is annotated as a reference next to the baseline bar.

the algorithms used in the OpenCL versions, *and that was used for both vector and GPU experiments*.

We use the largest available input for each benchmark, and each data point we report is an average of ten runs.

5.2 PORTABILITY AND COMPARISON WITH HAND TUNING

Figures 5.1a and 5.1b show the execution time of these applications on GPU and vector hardware respectively, normalized to the baselines mentioned above. Each bar shows segments for the time spent in the compute kernel (Kernel), copying data (Copy) and remaining time on the host. The total execution time for the baseline is shown above the bar.

Compared to the GPU baseline, HPVM achieves near hand-tuned OpenCL performance for all benchmark except bfs, where HPVM takes 22% longer. The overhead is because our

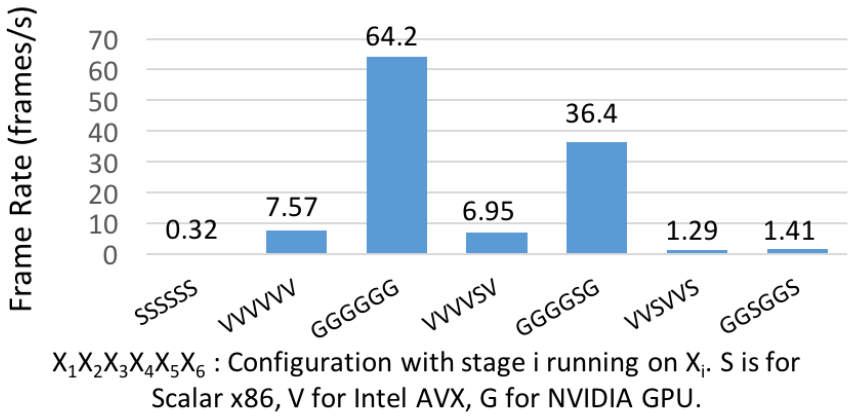


Figure 5.2: Frame rates of different configurations of Edge Detection six stage pipeline through single HPVM object code.

translator is not mature enough to generate global barriers on GPU, and thus HPVM version is based on a less optimized algorithm that issues more kernels than the `opencl_nvidia` version, incurring significant overhead.

In the vector case, HPVM achieves performance close to the hand-tuned baseline in all benchmarks except lbm. In this case, the vector code generated from the Intel OpenCL compiler after our SPIR backend is significantly worse that the one generated directly from OpenCL. We have not identified the cause of this yet.

Note that although HPVM is a low-level representation, it requires less information to achieve performance on par with OpenCL, e.g., details of data movement need not be specified, nor distinct command queues for independent kernels. The omitted details can be decided by the compiler, scheduler, and runtime instead of the programmer.

5.3 EVALUATION OF FLEXIBLE SCHEDULING

We used a six-stage image processing pipeline, Edge Detection in grey scale images, to evaluate the flexibility that HPVM provides in scheduling the execution of programs consisting of many dataflow nodes. The application accepts a stream of grey scale images, I, and a fixed mask B and computes a stream of binary images, E, that represent the edges of I. We feed 1280x1280 pixel frames from a video as the input and measure the frame rate at the output. This pipeline is natural to express in HPVM. The streaming edges and pipeline stages simply map to key features of HPVM, and the representation is similar to the code presented in Figure 3.1. In contrast, expressing pipelined streaming parallelism in OpenCL, PTX, SPIR-V or HSAIL, although possible, is extremely awkward, as explained briefly in

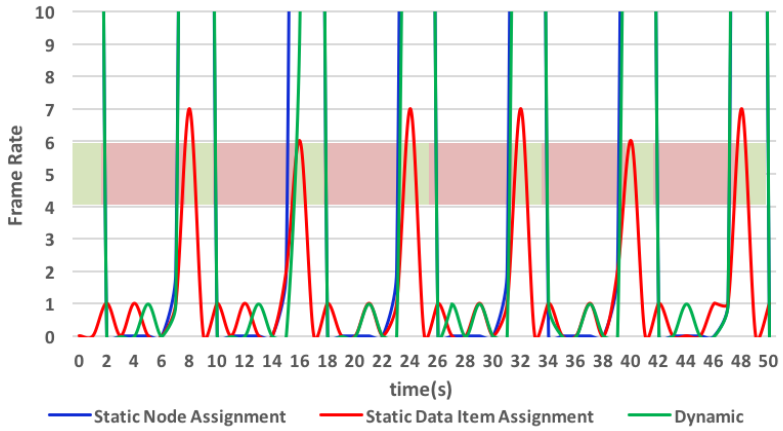


Figure 5.3: Edge Detection Frame rate with different scheduling policies. The green and red band in the graph indicates when the GPU is available or not respectively.

section 2.1.

Expressing this example in HPVM allows for flexibly mapping each stage to one of three targets (GPU, vector or a CPU thread), for a total of $3^6 = 729$ configurations, all generated from a single HPVM code. Figure 5.2 shows the frame rate of 7 such configurations. The figure shows that HPVM can capture pipelined, streaming computations effectively with good speedups. More importantly, however, the experiment shows that HPVM is flexible enough to allow a wide range of *static* mapping configurations with very different performance characteristics from a single code.

To show the flexibility for *dynamic* scheduling, we emulate a situation where the GPU becomes temporarily unavailable, by using a thread to toggle a boolean variable indicating availability. This can arise, e.g., for energy conservation in mobile devices, or if a rendering task arrives with higher priority. When the GPU becomes unavailable, kernels that have already been issued will run to completion but no new jobs can be submitted to it. We choose to have the GPU available for intervals of 2 seconds out of every 8 seconds, because the GPU in our system is far faster than the CPU.

In this environment, we execute the Edge Detection pipeline using the three different scheduling policies described in section 4.3.

Figure 5.3 shows the *instantaneous frame rate* for each policy. Green and red sections show when the GPU is available or not respectively. We truncate the Y-axis because the interesting behavior is at lower frame rates; the suppressed peak rates are about 64 frame/s.

Static node assignment policy makes no progress during the intervals when the GPU is not available. The other two policies are able to adapt and make progress even when the GPU is unavailable, though neither is perfect. Static data item assignment policy may or

may not continue executing when the GPU is unavailable, depending on when the data items that will be issued to the GPU are processed. Also, it may have low frame rate when the GPU is available, if data items that should be processed by the CPU execute while the GPU is available. Dynamic policy will not start using the GPU to execute a dataflow node for a data item until the node is done for the previous data item. That is why the frame rate does not immediately increase to the maximum when the GPU becomes available. The experiment shows HPVM enables flexible scheduling policies that can take advantage of static and dynamic information, and these policies are easy to implement directly on the HPVM graph representation.

We also used the Edge Detection code to evaluate the overhead of the scheduling mechanism. We compared the static node assignment policy using the runtime mechanism with the same node assignment using only compiler hints. The overheads were negligible.

Overall, these experiments show that HPVM enables flexible scheduling policies directly using the dataflow graphs.

5.4 NODE FUSION OPTIMIZATION EVALUATION

We evaluated the benefits of Node Fusion using two widely used kernels, Laplacian Estimate (L) and Gradient Computation (G). Most benchmarks we examined have been hand-tuned to apply such transformations manually, making it hard to find Node Fusion opportunities (although they may often be more natural to write without manual node fusion). The two kernels' dataflow graphs have similar structure, shown for L in Figure 3.1. We compiled the codes to run entirely on GPU and fed the same video frames as before. Fusing just the two independent nodes gave a speedup of 3.7% and 12.8% on L and G respectively. Fusing all three nodes yielded a speedup of 10.6% and 30.8% on L and G respectively. These experiments show that Node Fusion can yield significant gains, but the benefits depend heavily on which nodes are fused.

5.5 CONCLUSION

Experimental results show that HPVM optimizations achieve significant performance improvements, HPVM translators achieve performance competitive with manually developed OpenCL code for both GPUs and vector hardware, and that runtime scheduling policies can make use of both program and runtime information to exploit the flexible compilation capabilities.

Overall, we conclude that the HPVM representation is a promising basis for achieving performance portability and for implementing parallelizing compilers for heterogeneous parallel systems.

CHAPTER 6: APPROXHPVM

ApproxHPVM is a compiler IR and system designed to enable accuracy-aware performance and energy tuning on heterogeneous systems with multiple compute units and approximation methods. ApproxHPVM automatically translates end-to-end application-level quality metrics into accuracy requirements for individual operations. ApproxHPVM uses a hardware-agnostic accuracy-tuning phase to do this translation that provides greater portability across heterogeneous hardware platforms, with hardware specific decisions being deferred to install time.

ApproxHPVM incorporates three main components:

1. a compiler IR with hardware-agnostic approximation metrics. The ApproxHPVM IR is an extension of the HPVM IR (section 3).
2. a hardware-agnostic accuracy-tuning phase to identify error-tolerant computations.
3. an accuracy-aware hardware scheduler that maps error-tolerant computations to approximate hardware components.

As ApproxHPVM does not incorporate any hardware-specific knowledge as part of the IR, it can serve as a portable virtual ISA that can be shipped to all kinds of hardware platforms.

This chapter briefly describes the ApproxHPVM system [115]. It provides a high level work flow of the ApproxHPVM framework and mainly focuses on this thesis' contributions to this work, namely the design of the ApproxHPVM IR and the extensions on top of HPVM compiler infrastructure. The reader is referred to [115] and to the lead author, Hashim Sharif's, thesis for a full presentation of this work.

6.1 THE APPROXHPVM INTERMEDIATE REPRESENTATION

ApproxHPVM is inspired by and builds on HPVM [106]. We extend the HPVM IR in two key ways:

- to support execution of basic linear algebra tensor computations.
- to specify accuracy metrics for each operation.

6.1.1 Tensor operations in ApproxHPVM

Domain-specific languages such as TensorFlow and Pytorch allow for improved program-

Tensor Intrinsic	*Description*
i8* **llvm.hpvm.tensor.mul**(i8* lhs, i8* rhs)	Performs a matrix multiply operation on the input tensors.
i8* **llvm.hpvm.tensor.conv**(i8* input, i8* filter, i32 stride, i32 padding)	Applies a convolution filter on input tensor with configurable stride and padding.
i8* **llvm.hpvm.tensor.depthwise_conv**(i8* input, i8* filter, i32 stride, i32 padding)	Performs a depthwise tensor convolution with configurable stride and padding.
i8* **llvm.hpvm.tensor.add**(i8* lhs, i8* rhs)	Element-wise addition on input tensors.
i8* **llvm.hpvm.tensor.reduce_window**(i8* input, i32 reduction_type, i32 window_size)	Performs a (configurable) reduction operation over a specified window size on the input tensor.
i8* **llvm.hpvm.tensor.relu**(i8* input)	Element-wise relu activation function.
i8* **llvm.hpvm.tensor.clipped.relu**(i8* input)	Element-wise clipped relu activation function.
i8* **llvm.hpvm.tensor.tanh**(i8* input)	Element-wise tanh activation function.
i8* **llvm.hpvm.tensor.batchnorm**(i8* input, float* mean, float* variance)	Batch normalization on input tensor given mean and variance of batch.
i8* **llvm.hpvm.tensor.softmax**(i8* input)	Tensor Softmax.

Table 6.1: Tensor intrinsics in the ApproxHPVM representation.

mer productivity and are thus gaining wide-spread adoption. Accordingly, compilers such as XLA for TensorFlow [77] and TVM for MxNet [75] are beginning to support efficient mapping of high-level domain-specific abstractions to heterogeneous parallel compute units including CPUs, GPUs, FPGAs, and special-purpose accelerators, and to run-time libraries like cuDNN or cuBLAS.

A general-purpose parallel IR such as HPVM translates high-level operations into generic low-level LLVM instructions. However, such early lowering of domain-specific operations can result in loss of important semantic information that may be needed by a back end to target run-time libraries or domain-specific accelerators. Reconstructing the higher-level semantics after lowering is generally very difficult and sometimes infeasible.

Instead, we choose to incorporate high-level but broadly applicable operations into HPVM IR directly. In this work, we extend the HPVM IR representation with linear algebra tensor operations that allow for naturally expressing tensor-based applications. Tensors are used in a wide range of important domains, including mechanics, electromagnetics, theoretical physics, quantum computing, image processing and machine learning. For instance, convolutional neural networks may be expressed using generic linear-algebra operations. This design choice provides two essential benefits: a) It enables efficient mapping of tensor operations to special purpose hardware and highly optimized target-specific runtime libraries, such as cuDNN for GPUs. b) It allows approximation analyses to leverage domain-specific information, because the approximation properties, parameters, and analysis techniques usually are determined by properties of the domain.

The list of tensor intrinsics introduced in ApproxHPVM are listed in Table 6.1. The tensor

operations in ApproxHPVM are represented as calls to LLVM intrinsic functions (the same approach used within the HPVM compiler infrastructure, as described in section 4.1). The intrinsic calls appear to existing LLVM passes as calls to unknown external functions, so existing passes remain correct. For applications where all data-parallelism occurs via the tensor operations, the dataflow graph is only used to capture pipelined and task parallelism across nodes, while data-parallelism is captured by the tensor operation(s) within individual nodes.

Figure 6.1 shows a single neural network convolution layer represented in ApproxHPVM, using three tensor operations: `llvm.hpvm.tensor.conv`, `llvm.hpvm.tensor.add`, and `llvm.hpvm.tensor.relu`. The *DFG_root* function is the root of the dataflow graph, and would be invoked by host code. The root node is an internal graph node, which creates the leaf nodes *tensorConvNode*, *tensorAddNode* and *tensorTanhNode* (using `llvm.hpvm.createNode` calls) and connects the nodes through dataflow edges (using `llvm.hpvm.createEdge` calls). The leaf nodes invoke the tensor intrinsics to perform tensor computations on the input tensors. The output of the last node in the dataflow graph is connected to the output of the root node and is returned back to the caller.

6.1.2 Approximation Metrics in the IR

The second key feature of ApproxHPVM is the use of hardware-independent approximation metrics that quantify the accuracy of unreliable and approximate computations. We attach error metrics, defined below, as additional arguments to high-level tensor operations. Our design allows the specifications to be added to generic low-level instructions, but we do not use that in this work. To express the (allowable) difference between approximate and exact tensor outputs, we use vector distance metrics:

- Relative L_1 error:

$$L_1^e = \frac{L_1(A - G)}{L_1(G)} \tag{6.1}$$

  where

$$L_1(X) = \|X\|_1 = \sum_i |x_i| \tag{6.2}$$

  The numerator captures the sum of absolute differences between the approximate tensor output A and the golden tensor output G. The denominator is the L_1 norm of the golden output tensor, so that the ratio is the relative error.

```
define i8* @tensorConvNode(i8* %input, i8* %filter) {
  %result = call i8* @llvm.hpvm.tensor.conv(i8* %input, i8* %filter, i32* %strides, i32* %padding)
  return i8* %result
}
define i8* @tensorAddNode(i8* %input, i8* %bias_weights) {
  %result = call i8* @llvm.hpvm.tensor.add(i8* %input, i8* %bias_weights)
  return i8* %result
}
define i8* @tensorReluNode(i8* %input) {
  %result = call i8* @llvm.hpvm.tensor.relu(i8* %input)
  return i8* %result
}
define void @DFG_root(i8* %W, i8* %X, i8* %B) { ; Root node of the Dataflow Graph
  ; Creating DFG nodes
  %nodeConv = call i8* @llvm.hpvm.createNode(i8* @tensorConvNode)
  %nodeAdd = call i8* @llvm.hpvm.createNode(i8* @tensorAddNode)
  %nodeRelu = call i8* @llvm.hpvm.createNode(i8* @tensorReluNode)
  ; Creating data-flow edges between different DFG nodes
  call void @llvm.hpvm.createEdge(i8* %nodeConv, i8* %nodeAdd, 1, 0, 0, 0)
  call void @llvm.hpvm.createEdge(i8* %nodeAdd, i8* %nodeRelu, 1, 0, 0, 0)
  ; Binding the parent input to inputs of the leaf nodes
  call void @llvm.hpvm.bind.input(i8* %nodeConv, 0, 0, 0)
  call void @llvm.hpvm.bind.input(i8* %nodConv, 1, 1, 0)
  call void @llvm.hpvm.bind.input(i8* %nodeAdd, 2, 1, 0)
  ; Binding final DFG node output to parent node output
  call void @llvm.hpvm.bind.output(i8* %nodeRelu, 0, 0, 0)
}
```

Figure 6.1: Single Convolution Layer represented using ApproxHPVM tensor intrinsics. Convolution Layer sub-operations are represented as ApproxHPVM tensor intrinsics in HPVM dataflow nodes. The data-flow nodes are connected through explicit dataflow edges using HPVM intrinsics.

- Relative L_2 error:

$$L_2^e = \frac{L_2(A - G)}{L_2(G)} \tag{6.3}$$

  where

$$L_2(X) = \|X\|_2 = \sqrt{\sum_i x_i^2} \tag{6.4}$$

  This is similar to the L_1^e norm, except that the numerator represents the Euclidean distance and the denominator uses the L_2 norm.

Note that the relative L_1 error and relative L_2 error are non-negative and lie in the range $[0, +\infty)$.

Figure 6.2 shows how the approximation metrics are represented in the compiler IR. The two approximation parameters for each tensor operation are attached as additional arguments to the respective intrinsic functions. While our current system only uses the two metrics described, our implementation and analyses can be easily extended to include additional approximation metrics.

```
define i8* @tensorConvNode(i8* %input, i8* %filter) {
  %result = call i8* @llvm.hpvm.tensor.conv(i8* %input, i8* %filter, i32* %strides, i32* %padding, float
       %relative_l1, float %relative_l2)
  return i8* %result
}
define i8* @tensorAddNode(i8* %input, i8* %bias_tensor) {
  %result = call i8* @llvm.hpvm.tensor.add(i8* %input, i8* %bias_tensor, float %relative_l1, float
       %relative_l2)
  return i8* %result
}
define i8* @tensorReluNode(i8* %input) {
  %result = call i8* @llvm.hpvm.tensor.relu(i8* %input, float %relative_l1, float %relative_l2)
  return i8* %result
}
```

Figure 6.2: Convolution Layer represented using ApproxHPVM tensor intrinsics. Tensor intrinsics are annotated with accuracy metrics. The accuracy metrics L_1^e and L_2^e are passed as parameters to the intrinsic calls.

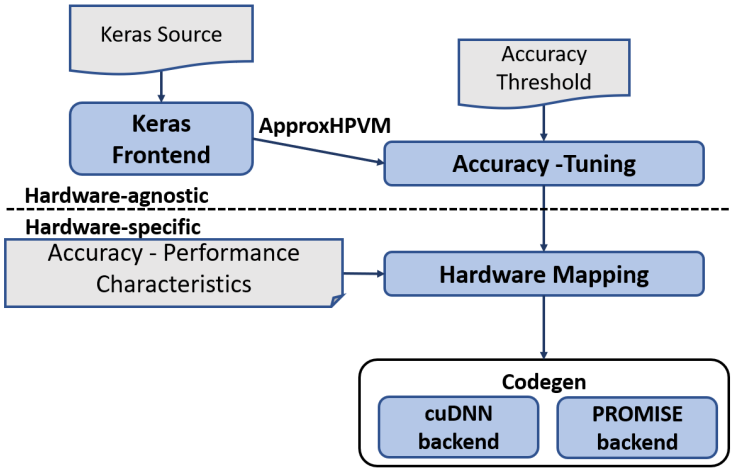


Figure 6.3: ApproxHPVM System Workflow

6.2 SYSTEM WORKFLOW

Figure 6.3 shows the overall ApproxHPVM workflow. The primary input is a program written using high-level abstractions of the Keras library [43], a popular open-source library for deep neural networks on TensorFlow. Our frontend translates a Keras source program to the ApproxHPVM IR. The second input is the programmer-specified end-to-end quality metric, a domain-dependent parameter. For the neural network domain, we use the acceptable loss in final classification accuracy and for image processing pipelines, we use desired Peak Signal-to-Noise Ratio (PSNR) of the approximated output.

The overall goal is to map the computations of the program to the compute units on a

target system, along with selected approximation parameter values on each compute unit, so that the program outputs satisfy the specified end-to-end accuracy. We decompose this mapping problem into a hardware-agnostic first stage and a hardware-specific second stage.

(1) The decomposition generates hardware-agnostic ApproxHPVM code with accuracy constraints, which enables portability of the ApproxHPVM virtual object code. As the accuracy constraints are hardware-independent, the IR operations can be mapped to a range of different types of hardware components with varying approximation choices. (2) The second stage is extremely fast, enabling techniques like dynamic accuracy-aware scheduling and rapid accuracy-aware hardware design space exploration, which would be impractical if the expensive first stage were needed.

The hardware-agnostic accuracy-tuning phase takes an end-to-end quality metric and computes the error tolerance for individual ApproxHPVM operations, adding these requirements in the IR. This phase guarantees that if these error tolerances for individual operations are (independently) satisfied, then the end-to-end accuracy specification will also be satisfied with some high probability, e.g., 95%.

The heart of the accuracy-tuner is an autotuning search that uses statistical error injection to model potential run-time errors and directly executes the program on a standard GPU to measure the end-to-end accuracy vs. the expected ("golden") output. The reasoning behind this choice is that maintaining the hardware agnostic aspect requires a hardware agnostic error model, a general error injection technique instead of a specific approximation choice and a way to evaluate which configuration has better performance. The autotuner is built using OpenTuner [78], an extensible framework for building domain-specific autotuners. The output of this stage is hardware-agnostic ApproxHPVM code, which is legal LLVM and can optionally be used as a virtual instruction set to ship the code as "virtual object code" to one or more targets [107].

For each target, a (static) accuracy-aware hardware mapping phase chooses which compute units should execute each tensor operation, and optimizes any approximation parameters available on each compute unit to minimize energy and/or maximize performance, while satisfying the *individual accuracy constraints on each operation.*

Finally, the code generation phase 6.3 leverages the hardware-specific backends to generate code for each compute unit.

6.3 CODE GENERATION

In its final phase, the ApproxHPVM compiler generates code for each operation corresponding to the selected compute unit.

Once we have the mapping of each operation in ApproxHPVM to the hardware unit and the hardware knob, we need to generate an executable that runs each operation on the respective hardware platform. We leverage the existing HPVM infrastructure for this.

We added two new backends: for PROMISE, targeting a library that performs optimized tensor computations on the PROMISE hardware simulator, and for GPU, targeting an optimized cuDNN and cuBLAS based library runtime. Since the support for backends is flexible, it can be extended to other approximate computing hardware platforms.

The back-end code generators translate dataflow graph nodes (containing tensor intrinsics such as `llvm.hpvm.tensor.mul`, `llvm.hpvm.tensor.conv`, etc) to functions that invoke the corresponding DNN operations for GPU or PROMISE.

6.3.1 PROMISE Backend

The PROMISE ISA exposes primitive instructions, at the level of vector add, vector dot product, etc, described in more detail in [19]. The PROMISE hardware simulator utilizes the ISA to implement a library of neural network layers as a higher level interface for PROMISE, which we target in this work.

Code generation for PROMISE requires an additional pattern-driven fusion operation. That is because translators within the HPVM infrastructure in principle traverse the dataflow graph and operate at the dataflow node level, while PROMISE operates on a granularity that is larger than that of a single dataflow node. As described, PROMISE can perform an entire layer operation as a single operation. In the ApproxHPVM representation, a layer operation is represented by one or more dataflow nodes, as shown for a convolution layer in figure 6.4.

A layer operation in a DNN usually maps to the following common patterns for *fully-connected* and *convolution* layers (equations 6.5 and 6.6 respectively):

$$Y_{FC} = f(X \cdot W + B) \tag{6.5}$$

$$Y_{Conv} = f(X \circledast W + B) \tag{6.6}$$

where W, X and B are the weight tensor, input tensor, and bias tensor, and $f(\cdot)$ is the activation function (sigmoid, relu, tanh, etc.). The PROMISE hardware simulator exposes the two following interfaces: `FCLayer_PROMISE` and `ConvLayer_PROMISE`, that implement the functionality of these layers on PROMISE.

We implement a pattern-driven Node Fusion transformation. The Node Fusion transformation operates on ApproxHPVM graphs. We traverse the dataflow graph using a topolog-

```
define i8* @tensorConvNode(i8* %input, i8* %filter) {
  %result = call i8* @llvm.hpvm.tensor.conv(i8* %input, i8* %filter, i32* %strides, i32* %padding)
  return i8* %result
}
define i8* @tensorAddNode(i8* %input, i8* %bias_weights) {
  %result = call i8* @llvm.hpvm.tensor.add(i8* %input, i8* %bias_weights)
  return i8* %result
}
define i8* @tensorReluNode(i8* %input) {
  %result = call i8* @llvm.hpvm.tensor.relu(i8* %input)
  return i8* %result
}
define void @DFG_root(i8* %W, i8* %X, i8* %B) { ; Root node of the Dataflow Graph
  ; Creating DFG nodes
  %nodeConv = call i8* @llvm.hpvm.createNode(i8* @tensorConvNode)
  %nodeAdd = call i8* @llvm.hpvm.createNode(i8* @tensorAddNode)
  %nodeRelu = call i8* @llvm.hpvm.createNode(i8* @tensorReluNode)
  ; Creating data-flow edges between different DFG nodes
  call void @llvm.hpvm.createEdge(i8* %nodeConv, i8* %nodeAdd, 1, 0, 0, 0)
  call void @llvm.hpvm.createEdge(i8* %nodeAdd, i8* %nodeRelu, 1, 0, 0, 0)
  ; Binding the parent input to inputs of the leaf nodes
  call void @llvm.hpvm.bind.input(i8* %nodeConv, 0, 0, 0)
  call void @llvm.hpvm.bind.input(i8* %nodConv, 1, 1, 0)
  call void @llvm.hpvm.bind.input(i8* %nodeAdd, 2, 1, 0)
  ; Binding final DFG node output to parent node output
  call void @llvm.hpvm.bind.output(i8* %nodeRelu, 0, 0, 0)
}
```

Figure 6.4: Convolution Layer in ApproxHPVM before Node Fusion.

ical sort, and identify *node sequences* that constitute fully connected or convolution layers. We refer to node sequence as a set of dataflow nodes that form a straight line graph. Each node must have a single predecessor (the previous node in the sequence) and a single successor (the next node in the sequence) except the start and end node that have no predecessor and successor in the sequence respectively. A node sequence corresponding to a fully connected layer is identified by a start node containing an `llvm.hpvm.tensor.mul` intrinsic, followed optionally by a node containing `llvm.hpvm.tensor.add` and an a node containing an activation intrinsic, in this order. Similarly, a convolution layer corresponds to a node sequence beginning with a node containing a `llvm.hpvm.tensor.conv`, optionally followed by `llvm.hpvm.tensor.add`, an activation intrinsic, and a pooling intrinsic, in this order.

Once an appropriate node sequence has been identified, if all nodes that belong to the node sequence have been annotated to be mapped to PROMISE (as enabled by the HPVM infrastructure), we proceed to the fusion of the nodes in the sequence. This involves the following main steps:

- Creating a new leaf node `FN` that eventually replaces the nodes in the layer node sequence. This involves creating a function that invokes the ApproxHPVM intrinsics corresponding to the operations performed by the identified layer. Dataflow

```
define i8* @convLayerNode(i8* %input, i8* %filter, i8* %bias_weights) {
  %conv_result = call i8* @llvm.hpvm.tensor.conv(i8* %input, i8* %filter, i32* %strides, i32* %padding)
  %add_result = call i8* @llvm.hpvm.tensor.add(i8* %conv_result, i8* %bias_weights)
  %relu_result = call i8* @llvm.hpvm.tensor.relu(i8* %add_result)
  return i8* %relu_result
}
define void @DFG_root(i8* %W, i8* %X, i8* %B) { ; Root node of the Dataflow Graph
  ; Creating DFG nodes
  %nodeConvLayer = call i8* @llvm.hpvm.createNode(i8* @convLayerNode)
  %nodeConv = call i8* @llvm.hpvm.createNode(i8* @tensorConvNode)
  %nodeAdd = call i8* @llvm.hpvm.createNode(i8* @tensorAddNode)
  %nodeRelu = call i8* @llvm.hpvm.createNode(i8* @tensorReluNode)
  ; Binding the parent input to inputs of the leaf nodes
  call void @llvm.hpvm.bind.input(i8* %nodeConvLayer, 0, 0, 0)
  call void @llvm.hpvm.bind.input(i8* %nodeConvLayer, 1, 1, 0)
  call void @llvm.hpvm.bind.input(i8* %nodeConvLayer, 2, 2, 0)
  ; Binding final DFG node output to parent node output
  call void @llvm.hpvm.bind.output(i8* %nodeConvLayer, 0, 0, 0)
}
```

Figure 6.5: Convolution Layer in ApproxHPVM after Node Fusion.

edges that were between nodes within the node sequence now express intra-node communication, and will now be expressed using registers. This is illustrated in figures 6.4 and 6.5, where the two edges of the root node between `nodeConv-nodeAdd` and `nodeAdd-nodeRelu` are simply represented by using registers `%conv_result` and `%add_result` in the body of the leaf node function convLayerNode. Dataflow edges to or from this node that are with nodes that do not belong in the node sequence represent data transfers that still need to be expressed at the dataflow graph level, and are handled at the next step.

- Updating the internal node that contained the node sequence, to contain the node `FN` instead. The internal node creates an instance of `FN`. Dataflow edges between the nodes of the identified sequence are deleted. Remaining edges/binds incoming to the start node of the node sequence are moved to the new node `FN`, and outgoing edges/binds of the end node of the sequence are moved to start from `FN`. The unused nodes are deleted.

Node Fusion is invoked before all code generation passes. This is safe to do, since it will only fuse node sequences where all nodes have been mapped to PROMISE. Therefore code generation for node sequences not mapped to PROMISE will not be transformed by Node Fusion. Instead, nodes belonging to such sequences will be handled by other backends.

After Node Fusion is complete, the granularity of the leaf dataflow nodes has been raised to that of layer operations, as supported by PROMISE. We simply map the instruction sequences found to the interfaces exposed by the PROMISE hardware simulator.

6.3.2 GPU Backend

Except from mapping the tensor operation of the ApproxHPVM dataflow nodes to the corresponding DNN operation, code generation for GPU requires an additional analysis pass to determine the legality of the translation.

Specifically, the ApproxHPVM tensor intrinsics have not been defined as in place operations, i.e. we expect them to always return a new tensor as the result of operating on their respective data. However, cuDNN and cuBLAS libraries only offer in place implementations for some of the tensor operations, updating one of their operands.

Therefore, we implement an analysis pass on top of ApproxHPVM dataflow graph, to determine when it is legal to use an in place implementation of a not in place operation, and we query this result during code generation for GPU.

Specifically, the analysis pass traverses the ApproxHPVM dataflow graph and determines for each dataflow node's input edges, whether the transferred operand can be used in an in place implementation. This is legal if both properties hold:

- It is defined as an output of a tensor intrinsic, not an argument passed to the dataflow graph by the host. This is expected, as the host code does not expect the values it passed to the dataflow graph to be changed unpredictably.
- It is only transferred by a single dataflow edge E. This means that that its value is expected to only be read by the destination node of E and then discarded, therefore it can be legally written after read within the destination node.

After this analysis is completed, we proceed to code generation. At a high level, this is a mapping between the ApproxHPVM intrinsic within each leaf node and the corresponding API call in our cuDNN and cuBLAS based library runtime. The backend consults the in place analysis when performing the translation, to ensure that the use of not in place operands within an intrinsic is valid when using an in place implementation of the intrinsic. The intrinsic is then replaced with the corresponding call to our library that implements its functionality.

The implementation of each call provides a wrapper around a base cuDNN or cuBLAS routine that performs the main computation. For the list of supported intrinsics, Table 6.2 indicates the corresponding cuDNN/cuBLAS call.

Tensor Intrinsic	*Base cuDNN/cuBLAS call*
i8* **llvm.hpvm.tensor.mul**	cublasSgemm
i8* **llvm.hpvm.tensor.conv**	cudnnConvolutionForward
i8* **llvm.hpvm.tensor.depthwise_conv**	cudnnConvolutionForward / Custom GPU kernel
i8* **llvm.hpvm.tensor.add**	cudnnAddTensor
i8* **llvm.hpvm.tensor.reduce_window**	cudnnPoolingForward
i8* **llvm.hpvm.tensor.relu**	cudnnActivationForward
i8* **llvm.hpvm.tensor.clipped.relu**	cudnnActivationForward
i8* **llvm.hpvm.tensor.tanh**	cudnnActivationForward
i8* **llvm.hpvm.tensor.batchnorm**	cudnnBatchNormalizationForwardInference
i8* **llvm.hpvm.tensor.softmax**	cudnnSoftmaxForward

Table 6.2: Corresponding cuDNN/cuBLAS calls for tensor intrinsics in the ApproxHPVM runtime.

6.4 EVALUATION

6.4.1 Platform

For our evaluation, we use the NVIDIA Jetson Tegra TX2 developer kit [116]. For the experiments done on PROMISE, we use the functional simulator and the timing and energy model obtained from its authors [19]. The SOC we model extends the Tegra TX2 board with the PROMISE accelerator. The communication across the GPU, CPU, and PROMISE happens through main memory.

We utilize a split approach where we collect profiler performance and energy numbers from direct execution on the GPU and simulator results for operations mapped to PROMISE.

6.4.2 Functional Experiments Methodology

To verify the functional correctness of our generated binaries and to measure the end-to-end accuracy of each network given a configuration, we use the GPU in tandem with PROMISE functional simulator. If a layer is mapped to the GPU, the corresponding tensor operations are executed on the GPU. If a layer is mapped on PROMISE, it is offloaded to PROMISE functional simulator. Consequently, the final result is the same as it would be if these operations were all executed on a real SoC containing both a GPU and PROMISE. Since the PROMISE simulator adds Gaussian random error to each run, we use statistical testing to measure the fraction of program runs that satisfy the end-to-end quality metric - we call this $R_{success}$. We run each configuration 200 times to obtain the mean and standard deviation of the classification accuracy, and $R_{success}$ of the configuration.

6.4.3 Time and Energy Experiments Methodology

We measure the time and energy of tensor operations on GPU by assigning a timestamp pair to the beginning and end of each tensor operation and reading the Jetson's power rails at regular intervals for the GPU and DRAM power. Total time is computed by subtracting the timestamp difference, and total energy by integrating the power readings.

We obtain per-tensor operation time and energy for both full-precision floating point (FP32) and half-precision floating point (FP16) for each benchmark, using the average time and energy over 100 runs. Instead of rerunning an operation on the GPU each time we ran a configuration, we collected these results once per benchmark and tabulated them. Then, whenever a particular tensor operation or network layer was mapped to the GPU, we obtained the required values from this lookup table.

We use a timing and energy model for PROMISE, that analytically computes the time and energy of tensor operations on PROMISE, by accumulating the compute cost with the cost of loading data from and writing data to main memory.

We obtained the total time and energy for a network by summing the time and energy of each layer, using the PROMISE analytical model if the layer was mapped to PROMISE or the GPU time and energy lookup tables.

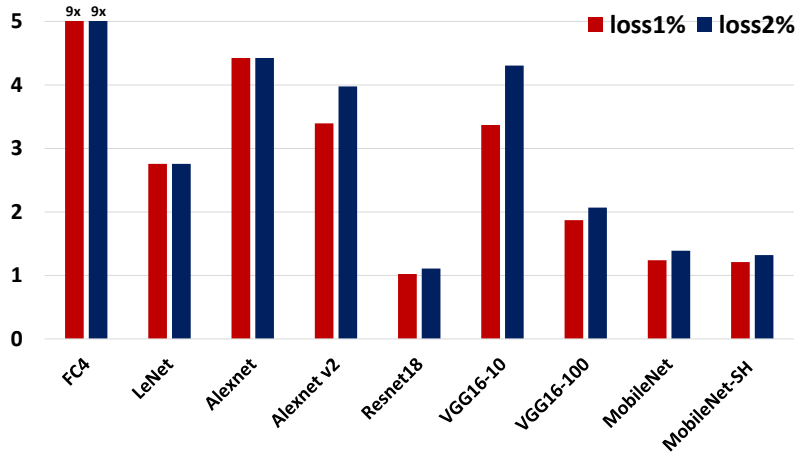
6.4.4 Benchmarks

We evaluate ApproxHPVM on nine benchmarks from the deep learning domain and five convolution-based image processing benchmarks.

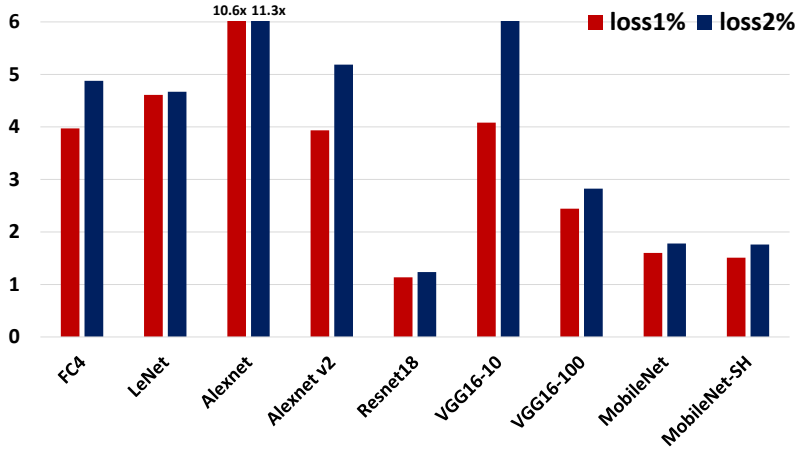
DNN Benchmarks. We use popular DNN benchmarks including LeNet[117], AlexNet [118] (reference implementation [119]), ResNet-18 [120], VGG-16 [121] on two different datasets, MobileNet [122], and Shallow MobileNet [122]. We created a variant of Alexnet (called Alexnet v2) that includes an extra convolution layer and provides approximately 6% higher end-to-end accuracy. We also include a 4 layer fully-connected DNN, called FC-4.

We use MNIST [123] (Lenet, FC-4), CIFAR-100 [124] (VGG-16), and CIFAR-10 (Alexnet, Alexnet v2, Resnet-18, VGG-16, MobileNet, Shallow MobileNet). All of them have 60K images, which we split into 50K for training and 10K for inference. The latter is split further into 5K calibration and validation sets for autotuning and evaluation respectively.

Image Processing Benchmarks. We also include 5 convolution-based image processing benchmarks. We construct these benchmarks by including different combinations of commonly-used image filters: Gaussian (G), Emboss (E), Outline (O), MotionBlur (M), and Sharpen (S). At the IR level, the filters are represented as tensor convolutions, with the



(a) Speedup



(b) Energy Reduction

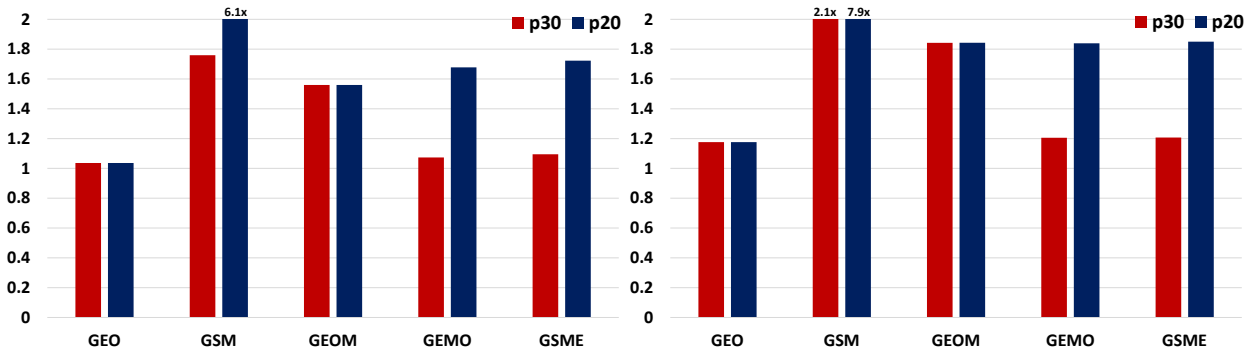
Figure 6.6: Speedup and energy reduction (over baseline) of all nine DNNs for $Loss_{1\%}$ and $Loss_{2\%}$ experiments (higher is better). 6.6a: Speedup. 6.6b: Energy reduction.

exception of Emboss which is a convolution followed by a bias add operation. The identified filter combinations are GEO, GSM, GEOM, GEMO, and GSME.

6.4.5 Results

Our results show that our framework can offload approximable computations to special-purpose accelerators that provide significant gains in performance and energy, while staying within user-specified application-level quality metrics with high probability. A less strict user-specified quality metric leads to more opportunities for approximation, allowing for higher performance and energy benefits on average.

Across the DNN benchmarks, we observe 1.02-9x performance speedups (figure 6.6a) and



(a) Speedup

(b) Energy Reduction

Figure 6.7: Speedup and energy reduction (over baseline) of all 5 image processing benchmarks for $PSNR_{30}$ (p30) and $PSNR_{20}$ (p20) thresholds. 6.7a: Speedup. 6.7b: Energy reduction.

1.14-11.3x energy reduction (figure 6.6b) with user-specified accuracy loss threshold of 1% or 2%. Most networks obtain from 1.5x–4x and 1.5x–5x improvements in performance and energy respectively.

Image processing benchmarks show performance and energy benefits ranging from 1.04x to 6.1x (figure 6.7a) and from 1.2x to 7.9x (figure 6.7b) respectively for user-specified application-level quality metric $PSNR_{30}$, with the benefits increasing when the quality specification dropped to $PSNR_{20}$.

CHAPTER 7: APPROXTUNER: A COMPILER AND RUNTIME SYSTEM FOR ADAPTIVE APPROXIMATIONS

With the increasing need for machine learning and data processing near the edge, software stacks and compilers must provide optimizations for alleviating the computational burden on low-end edge devices. Approximate computing can help bridge the gap between increasing computational demands and limited compute power on such devices.

We present ApproxTuner, a portable optimizing compiler and runtime system that enables flexible, optimized used of multiple software and hardware approximations in a unified easy-to-use framework. ApproxTuner provides a wide range of software and hardware approximations, focused on tensor operations used in deep neural networks and image processing. ApproxTuner uses a combination of development-time and install-time analyses to leverage both hardware-independent approximations and hardware-specific approximations, while preserving application portability. To reduce the cost of install-time autotuning, we propose a novel mechanism called federated autotuning for retuning with additional hardware-specific approximation choices. At runtime, ApproxTuner leverages a Pareto-optimal curve of approximation configurations constructed in the offline phases to dynamically tune approximation levels in response to load and power fluctuations.

Across 10 evaluated benchmarks from deep learning and image processing domains, ApproxTuner provides significant benefits. For 7 DNN benchmarks, ApproxTuner achieves geometric mean speedup of 1.9x and geometric mean energy reduction of 2x with only hardware-independent approximations. Additionally, install-time retuning with hardware-specific approximation choices further improves these results to a geometric mean speedup of 5.6x and geometric mean energy reduction of 5.9x. Similarly, for 3 image processing benchmarks, ApproxTuner achieves a geometric mean speedup of 2.14x and a geometric mean energy reduction of 2.4x.

7.1 MOTIVATING EXAMPLE

Section 1.4.3 describes, at a high level, our approach in selecting approximation techniques by decomposing the static selection, i.e. before dynamic approximation tuning, in two stages: ahead-of-time hardware-independent approximation tuning and install-time hardware-specific retuning. Both the ahead-of-time, development-time tuning and the install-time tuning utilize actual approximation techniques in the autotuning loop instead of artificial error injection and L1-error and L2-error metrics to capture its effect, as illustrated in ApproxHPVM (refer to section 6.2).

This section demonstrates that the previous approach is not general enough to capture a wide range of approximations. This is because each approximation technique and each approximation knob has its own unique error model. Hence, in ApproxTuner we use the actual approximation methods in the autotuning loop instead of an artificial error injection process that simulates a single error model.

To illustrate this, figure 7.1 shows 5 layers of the VGG-16 DNN and two potential mappings of approximation knobs to tensor operations. The first configuration applies *filter sampling* to all 5 layers and the second configuration maps 4 layers to *PROMISE*, the analog deep learning accelerator that we target throughout this work, as mentioned in 1.1.1. For each of the layers in both configurations, we measure the L1-norm and L2-norm (compared to the ground truth tensors) and also generate an error distribution. Note that layers in both configurations have a Gaussian error distribution. Moreover, the L1- and L2- norms of the first 4 layers in the filter-sampling based configuration are higher than the L1- and L2- norms of the corresponding layers in the PROMISE based configuration, suggesting that the end-to-end accuracy loss of the PROMISE based configuration should be lower. However, the PROMISE based configuration has a 2.3 percentage point higher accuracy loss (83.93%) compared to the filter-sampling based configuration (86.26%). This shows that using aggregate error metrics to capture the operation-level error distribution does not necessarily capture the actual sensitivity of the end-to-end application result.

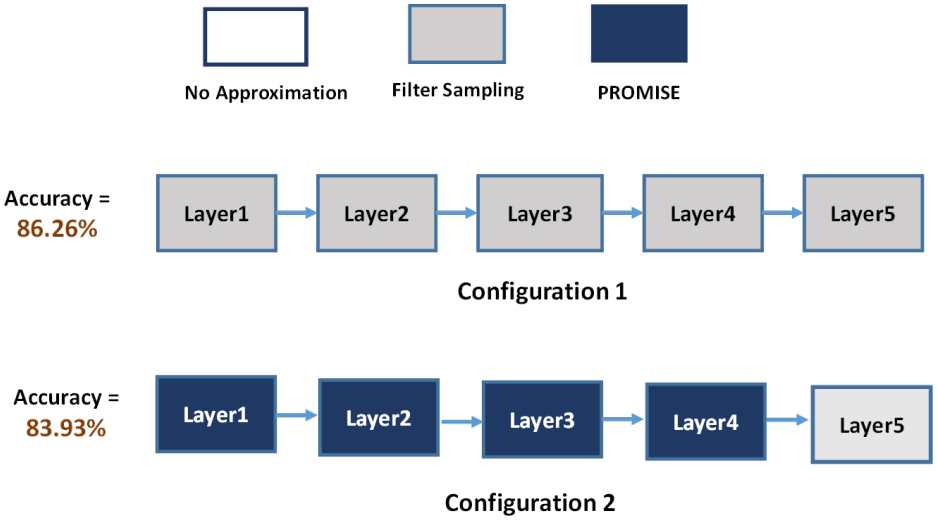


Figure 7.1: End-to-end accuracy impact of applying different Filter-Sampling vs PROMISE to layers in VGG-16.

7.2 APPROXTUNER

Figure 7.2 shows the high-level system workflow of ApproxTuner. Broadly, the system is composed of three phases: i) development-time, ii) install-time, and iii) runtime. ApproxTuner takes as input programs in the ApproxHPVM Intermediate Representation, that is HPVM IR extended with tensor intrinsics that represent data parallel patterns. These data parallel patterns include high-level linear algebra tensor operations used in the domains of neural networks and image processing. Tables 6.1 and 7.1 include the tensor intrinsics supported in the IR, for the domains of neural networks and image processing respectively.

The development-time phase includes an approximation tuner that takes as input a program and a domain-specific quality threshold. This phase is designed to enable ahead-of-time approximation tuning in an application-specific but hardware-independent manner. The approximation tuner uses an autotuning search to identify mappings of individual tensor operations to software approximations that maximize performance and energy benefits while staying within the provided end-to-end quality constraint. The output of this phase is a set of configurations that are then shipped along with the application. This phase is detailed in section 7.2.3. The approximation methods employed are detailed in section 7.2.2.

The install-time phase uses the configurations shipped from the development-time step and generates a performance-accuracy Pareto-optimal curve. Since the performance of software approximations is also hardware-dependent, the Pareto curve is constructed after profiling the different configurations on the target hardware. The install-time phase includes an optional *retuning* phase. The retuning phase attempts to maximize performance benefits by mapping computations to hardware-specific knobs for approximation. To render this feasible for low-end edge devices, we also propose a strategy called *federated autotuning* detailed in section 7.2.4.

At runtime, the performance-vs-accuracy Pareto curve constructed at install-time is used for dynamic approximation tuning. In the face of load, power, and frequency variations, the performance monitor gives feedback to the dynamic control. Based on the feedback, the dynamic control computes a target speedup to maintain a required level of system responsiveness (configurable by the user). The runtime control is described in section 7.2.5.

7.2.1 Tensor Operations

ApproxTuner is implemented on top of the HPVM compiler infrastructure. Therefore, to represent tensor-based domains such as DNNs and image processing (among potential others), we extend the tensor operations supported in ApproxHPVM (refer to table 6.1)

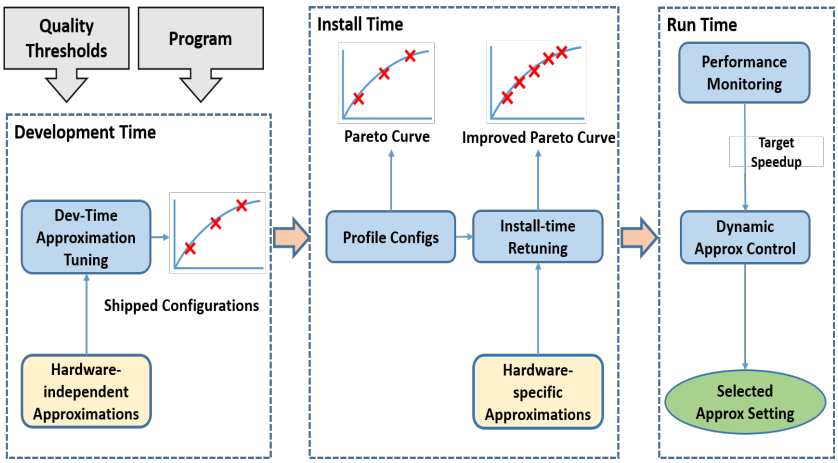


Figure 7.2: End-to-end system workflow of ApproxTuner.

Tensor Intrinsic	*Description*
i8* **llvm.hpvm.tensor.map**(i8* function, i8* input1, i8* input2, ...)	Zips multiple equal-shaped tensors and applies function element-wise.
i8* **llvm.hpvm.tensor.reduce**(i8* function, i8* input, i32 axis)	Performs a reduction operation along an axis of the input tensor.
i8* **llvm.hpvm.tensor.fft2D**(i8* input)	2-dimensional Fast Fourier Transform on the last 2 dimensions of the input tensor.

Table 7.1: Additional Tensor intrinsics in the ApproxHPVM representation

with additional tensor operations, identified after studying the image processing domain. Table 7.1 lists the tensor operations identified for the image processing domain. Section 6.1.1 describes the benefits of the design choice of including high level operations at the IR level, as well as the implementation mechanism of choice for them, the use of LLVM intrinsics.

7.2.2 Approximation Methods

We describe the approximation optimizations used in ApproxTuner. Specifically, we include 3 software approximations techniques and 2 hardware approximation techniques. The software approximation techniques include: i) filter sampling for convolutions, ii) perforated convolutions, and iii) reduction sampling. The hardware ones include i) reduced floating point precision (FP16) and ii) a highly energy and performance-efficient analog accelerator, PROMISE [19]. Note that our framework is extensible to all types of software and hardware approximations.

Filter Sampling for Convolutions. Convolutions are widely used in deep learning, but they are expensive to compute and therefore important to optimize [125]. Based on Kernel Perforation proposed by Maier et al. [126], we implement filter sampling, which skips certain

input and filter elements in the computation of each output element. For example, with a sampling rate of 50%, every other element in both the filter and the input are skipped. This technique shrinks the size of the filter and the input, reducing both compute operations and memory traffic.

Rescaling the values of the sampled filter elements is necessary for retaining reasonable end-to-end accuracy. For example, we found that rescaling reduces absolute end-to-end accuracy loss by 60% for ResNet-18 when using 25% filter sampling across all convolutional layers. Based on empirical testing, we apply a rescaling factor of $\frac{stride}{stride-1}$ to all filter elements.

We support two sampling rates for filter sampling: 50% (skip 1 out of 2) and 25% (skip 1 out of 4); benefits become negligible with lower sampling rates. The skip rate is exposed as a knob to the autotuner.

We present an additional knob to our autotuner: the initial offset at which the elements are skipped. We have observed this to have a noticeable impact on overall accuracy, as different offsets may align with more or less important filter elements. We permit an initial offset up to the length of the skipping stride.

Perforated Convolutions. Recent work [103] has proposed *perforated convolutions*, which correctly computes a subset of the spatial positions of the output tensor and computes the rest using nearest neighbor averaging from the previously computed positions. This technique reduces the size of the input matrix that feeds into the convolution thereby reducing both the data movement and the compute overheads.

Figurnov et al. [103] propose this technique and several strategies for selecting perforation patterns; we select row- and column-based perforation. Row perforation skip whole rows at a regular stride, and similarly column perforation skips columns. We export a *skip rate* as a knob to the autotuner, and supports 50% skip rate (skip 1 out of 2) and 33% skip rate (skip 1 out of 3). Perforating less than 33% does not provide any noticeable improvements, due to the overhead introduced when interpolating the perforated elements.

Similar to filter sampling, we have observed a sizable impact of the initial offset on the accuracy, and also expose the initial offset as a tuning knob.

Reduced Floating Point Precision. Reduced floating point precision is used in many domains including machine learning and image processing; it has been shown that reduced precision often has minimal accuracy impact while providing significant performance improvements [127].

We specifically target IEEE 754 half-precision floating point (FP16), which is supported across a variety of devices and deep learning frameworks. FP16 reduces memory bandwidth usage by up to 50% and can achieve great acceleration with modern GPU supports. We support FP16 for all the tensor operations in our framework.

Reduction Sampling. Tensor reduction reduces a number of elements along an axis into a single value. When reducing N elements into one, reduction sampling samples from these elements with a sampling ratio $\eta \leq 1$, so that only ηN elements are used in the computation. Zhu et al [97] propose reduction sampling and prove error bounds for several special cases. We implement this sampling technique and expose the sampling ratio as a knob to the autotuner.

Depending on the task performed by reduction, sampling can incur not only random but also systematic error on the result. For example, the result *in expectation* of a sampled average reduction is equal to that of a precise reduction, but a sampled summation needs to be scaled up by $\frac{1}{\eta}$ to remove systematic errors. For many common cases of reduction, such as summation, average, max/min, and argmax/argmin, we can apply a corresponding adjustment to compensate for this error.

Analog Computations with PROMISE accelerator. We support mapping to the PROMISE accelerator that provides efficient convolution and matrix multiply operations in the analog domain. PROMISE uses in-memory, low signal-to-noise ratio analog computation on the bit lines of an SRAM array to perform faster and energy efficient vector dot products. Srivastava et al. [19] show that PROMISE consumes 3.4-5.5x less energy and has 1.4-3.4x higher throughput than custom *non-programmable* digital accelerators. The PROMISE hardware provides seven different voltage levels (P1-P7), in increasing order of voltage and decreasing error.

7.2.3 Development-time Approximation Tuning

Prior to shipping application code, the development-time approximation tuner applies approximations that are not specific to any particular hardware configuration. The only hardware-based approximation choice we consider in this phase is reduced floating point precision (FP16) since it is supported on most modern GPUs and recent CPUs [128, 129, 130]. The goal of this phase is to identify mappings of approximations to ops that provide an optimal trade-off between performance and accuracy loss (i.e., maximize performance for given accuracy). Figure 7.3 shows the workflow for development-time approximation tuning.

Search Space of Approximations. In the autotuning search, we include 4 approximation techniques from section 7.2.2: perforated convolutions, filter sampling for convolutions, reduction sampling, and FP16 reduced precision. For each, we consider various knobs as shown in figure 7.4:

Note that the possible offsets for each skip rate differ. If the skip factor is 50% (skip 1 in 2) only two offsets are possible: {0,1}. Similarly for 33% perforation, 3 offsets are

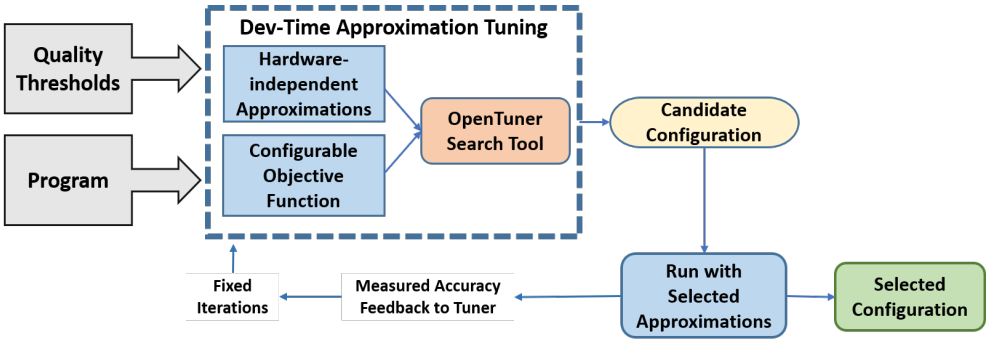


Figure 7.3: Development-time autotuning workflow.

```
perforation:  {row, col}  {50%, 33%}, offsets = {0, 1, 2}
filter_sampling:   {50%, 25%}, offsets = {0, 1, 2, 3}
reduction_sampling:  {50%, 40%, 25%}
FP precision: {FP32, FP16}
```

Figure 7.4: Knobs for approximation techniques.

possible and for 25% sampling, 4 offsets are possible. Hence, in total, we have 10 choices for perforation with 5 each for row-only and column-only perforation, each with varying offsets and perforation rates (33%, 50%). For sampling, we have 6 choices with varying skip offsets (2 for 50% sampling and 4 for 25% sampling). For reduction, we support 3 choices. For precision, we support 2 choices (FP16 and FP32).

Autotuning Workflow. To enable efficient search, we use OpenTuner [78], an extensible library that provides an API for developing autotuners with customizable search spaces and objective functions that rank configurations. We use OpenTuner since it has been shown to provide good results with search spaces as large as 10^{3600} possible configurations. To extract configurations with varying performance-accuracy tradeoffs, we tune with different accuracy thresholds; e.g., Loss $\leq$ 1%, Loss $\leq$ 2%, and Loss $\leq$ 3%. We run the autotuner separately for each accuracy threshold for a total of 3000 autotuning iterations.

Hardware-agnostic objective function. To drive the autotuner search over the configuration space, we use a hardware-independent objective function. The objective function uses operation count (i.e., the amount of work) as a proxy for the predicted execution time of a configuration. The speedup of a configuration is computed as the ratio of the cost of the baseline configuration (no approximations) and predicted execution time of the particular configuration. A configuration, *config*, is the mapping of an approximation choice, *K(i)*, to each tensor operation, *op(i)*. We denote the cost of mapping *K(i)* for *op(i)* as *C(op(i), K(i))*. We sum the costs for individual tensor operations to derive the cost function C_{Total}

of the full candidate configuration consisting of N total tensor operations as:

$$C_{Total}(config) = \sum_{i=0}^{N} C(op(i), K(i)) \tag{7.1}$$

For the cost (execution time) of running approximation K on operation op, we use a proxy based on the sum of the count of memory operations and compute operations:

$$C(op, K) = \frac{N_m(op)}{R_m(K)} + \frac{N_c(op)}{R_c(K)} \tag{7.2}$$

where N_m and N_c are the number of memory and compute operations, respectively, for the baseline (non-approximate) version of op. R_m and R_c are the corresponding reduction factors applied that are specific to each approximation method. For instance, an FP16 50% filter sampling method would have $R_m = 4$ since it loads 2× less memory due to FP16 and 2× less loads because only 50% of input and filter values are loaded from memory, $R_c = 2$ since it skips half the computations. This coarse heuristic estimation allows for assigning the configurations a relative ordering and drives the autotuning search towards more profitable configurations. That is, the search seeks candidate configurations that satisfy the accuracy threshold at the lowest cost based on the above heuristic.

Shipping Configurations. Once the autotuner has discovered the best configurations that satisfy each considered accuracy threshold, these configurations (with their associated speedup and accuracy loss) are shipped with the application package. To avoid shipping suboptimal configurations, we construct a Pareto-frontier using the predicted speedups and skip configurations that do not lie on the Pareto-frontier for speedup vs. accuracy. Across the 7 DNNs, we observe this to reduce the shipped configurations on average by 3.2x.

7.2.4 Install-time Federated Tuning

At install time, ApproxTuner performs two additional tuning tasks. First, it updates the performance estimates of the configurations shipped with the application by measuring the performance directly on the target hardware, in order to obtain a more precise Pareto-optimal curve. Second, it optionally invokes an install-time retuning phase that can exploit hardware-specific knobs for approximation to maximize performance and energy benefits.

The optional install-time retuning can be prohibitively expensive on resource-constrained edge compute devices (as our experiments show). Inspired by federated machine learning [131, 132], we propose federated autotuning, which renders install-time autotuning fea-

sible for resource constrained edge devices. Federated learning involves a centralized control that coordinates among distributed devices and combines the results of the individual distributed training runs. Similarly, federated autotuning distributes the autotuning task among edge devices where a single edge device performs a fraction of the autotuning and returns the top performing configurations back to the centralized server.

Client-server Interaction. At the heart of federated tuning is a centralized server that coordinates tuning among distributed edge devices. The client receives the development-time shipped configurations and the calibration inputs (used for retuning) from the server. The server has a database to cache the results retrieved from the edge devices. After the system converges on good configurations, edge devices can simply retrieve a Pareto-optimal curve for configuration performance vs. accuracy from the coordinating server, hence avoiding the need to do tuning on the edge device. Figure 7.6 shows the overall workflow of federated tuning and the interaction between the centralized cloud server and distributed edge devices.

Synergy between development-time and install-time tuning. While autotuning has been shown to work with significantly large search spaces [78], autotuning can require a large time budget. In ApproxTuner we exploit the synergy between the development-time phase and the install-time phase to accelerate autotuning at install-time by leveraging some information from the ahead-of-time approximation tuning phase. The install-time tuning phase is designed to utilize approximation sensitivity information being fed from the development phase tuning. For instance, the development-time phase may identify that a certain convolution layer is very sensitive to approximation. The retuning phase leverages such sensitivity information to reduce the search space of approximation choices to be considered for each operation. Since the application is shipped with a set of configuration points, analyzing the union of points allows for eliminating approximation choices on a per-operation basis. The install-time retuning process can be summarized as:

1. Scan all the shipped configurations to construct a list of approximation choices to be considered for each tensor operation. We simply take the union of all approximations that were mapped to a certain operation across all shipped configurations.

2. The search space of approximations must include the set of choices identified in step 1 and any additional hardware approximation choices to be considered. Operations that were not mapped to any approximation (in any of the configurations) are considered sensitive and are skipped for approximation in the retuning step.

3. Start the autotuning loop, where the autotuner considers only the selected subset of approximations above for a given operation.

```
Configuration: {
    tensor.conv: samp50%, samp25%, hardware_knobs,
    tensor.conv: perf33%, hardware_knobs,
    tensor.conv: fp32
}
```

Figure 7.5: Approximation choices per tensor operation.

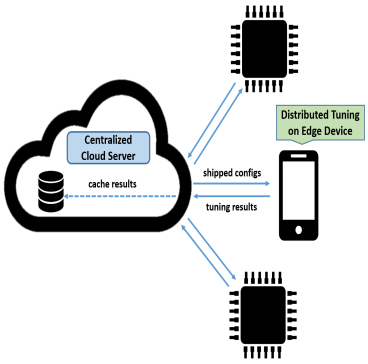


Figure 7.6: Federated Autotuning Workflow

Example. Consider a benchmark with 3 convolution operations with 20 different configurations shipped (not shown). After Step 1 above, for each tensor operation, we consider only the union of approximations that were mapped to that operation across all 20 configurations. Consider the following to be the space of approximation choices that were mapped to each operation, as shown in figure 7.5:

Note that no hardware knobs are allocated to the third convolution because it was deemed not approximable at development time (albeit for the software approximations considered then). For the target platform considered in our evaluation, we have observed this design choice to work well. We also allow this behaviour to be configurable for scenarios where the user would like to control how the hardware knobs get included/excluded from the search space.

7.2.5 Runtime Dynamic Approximation Tuning

Our dynamic tuning scenario is invoked when resources available to our application change dynamically. We assume our application runs in a loop, with each iteration performing inferences (for the DNN) or image processing for the images currently being considered. We assume an independent mechanism (e.g., a real-time scheduler or the operating system) that determines how much time (and energy) can be allocated to the next iteration of the

application and how much accuracy loss is acceptable. This decision must take into account other applications in the system, available resources (e.g., battery life and bandwidth), and an overall notion of utility for the entire system. Significant prior work has addressed this problem (e.g., [133]) and it is outside the scope of this work. This work concerns how ApproxTuner can determine the right configuration (approximations for the different tensor operations), given that the system allocates certain resources and accuracy requirements for this application.

Our dynamic tuning approach utilizes the Pareto curve for speedup vs. accuracy loss generated through the combination of the development time and install time tuning phases. Each point on the Pareto curve contains the program configuration, which includes the mapping of an approximation to a tensor operation that achieves the best speedup for the given accuracy. At runtime, whenever the application receives a new set of expectations from the system, it invokes an application level controller at the end of the application loop to determine the appropriate configuration choice from the Pareto curve for the next iteration. We implement different control strategies; e.g.,

1. *Achieve target speedup (resp. energy reduction, accuracy loss)*: Search in the Pareto-optimal set for the first configuration that achieves speedup (resp. energy reduction, accuracy loss) no lower than the target speedup (resp. energy reduction, accuracy loss), namely C. Select C.

2. *Achieve target speedup on average*: The first strategy overshoots the desired target speedup because the selected configuration in the Pareto-optimal set may deliver higher speedup, but this comes at the cost of potentially worse accuracy. Instead, we can search for the configuration, C, but switch to it with some probability, p_C, and otherwise remain with the previous configuration, with lower speedup but better accuracy. The probability p_C is chosen so that $E(p_C \cdot speedup_C + (1 - p_C) \cdot speedup_P) = target\_speedup$.

The different control strategies have different strengths and applications. Deadline driven systems, needing to ensure that the deadline is met at all times, would select strategy 1. Systems focused on maintaining the desired performance goal in the long term, in which it is acceptable for individual outputs to be sometimes slower than the target speedup, would benefit from strategy 2.

7.3 METHODOLOGY

7.3.1 Benchmarks

We evaluate 7 DNN benchmarks (Table 7.2a) and 3 image processing benchmarks (Table 7.2b).

Datasets. We evaluate our DNNs on 3 datasets: i) MNIST [123] for handwritten digits, ii) CIFAR-10 [124] with $3 \times 32 \times 32$ sized color images belonging to 10 classes, and iii) CIFAR-100 [124] with $3 \times 32 \times 32$ sized color images belonging to 100 distinct classes.

Each dataset has a total of 60K images, from which we randomly sample 10K into equal-sized calibration and test sets (5K each). The calibration set is used in the autotuning phases while the test set is used to evaluate the accuracy impact on unseen data.

For image processing benchmarks, we use the classification dataset from Intel Scene Classification Challenge [134] (referred to as "Scene" dataset below). This dataset contains 24K $3 \times 150 \times 150$ sized color images. We randomly sample 6K images from the dataset into equal-sized calibration and test set (3K each) for autotuning and test respectively.

DNN Benchmarks. We include popular DNN models in our evaluation including: Alexnet, Resnet-18, MobileNet, VGG-16, and Lenet-5 networks. We trained VGG-16 for both CIFAR-10 and CIFAR-100 since it provides reasonably good end-to-end accuracy on both datasets [121]. We also use a variant of Alexnet (called Alexnet2) that makes slight adaptations (including one additional convolution layer) to the original Alexnet model to give better end-to-end accuracy on the CIFAR-10 dataset. We observe this variant to provide an additional 6% higher end-to-end accuracy. We also include MobileNet which is known to be an efficient DNN model in terms of both performance and model size.

Image Processing Benchmarks. We also include 3 image processing benchmarks extracted from common image processing tasks: Blending, Blur, and Canny (Table 7.2b). As Blending is a binary kernel (i.e., takes two images and yields one image), we use both the dataset and a shuffled copy of the dataset as input. At the IR level, these kernels apply convolution, map and reduce operations.

7.3.2 Quality Metrics

For the DNN benchmarks, we measure the accuracy degradation with respect to the baseline. $Loss_{x\%}$ refers to an accuracy degradation of x percentage points compared to the baseline.

For the image processing benchmarks, we use the average PSNR to quantify the error in

Network	Dataset	Layers	Accuracy
Alexnet	CIFAR-10	6	79.16%
Alexnet2	CIFAR-10	7	85.09%
Resnet-18	CIFAR-10	22	89.44%
VGG-16-10	CIFAR-10	15	89.41%
VGG-16-100	CIFAR-100	15	66.19%
Lenet-5	MNIST	4	98.7%
MobileNet	CIFAR-10	28	83.69%

(a) DNN benchmarks, corresponding datasets, layer count, and classification accuracy with FP32 baseline.

Benchmark	Dataset	Description
Blending	Scene	Image sharpening + 2-image blending
Blur	Scene	Image blur with a Gaussian kernel
Canny	Scene	Canny edge detection

(b) Image processing benchmarks and corresponding datasets.

Table 7.2: Description of evaluated benchmarks.

Tegra TX2 Parameters	
CPU Cores	6
GPU SMs	2
GPU Cores	128
GPU Frequency	1.12 GHz
DRAM Size	8 GB
PROMISE Parameters	
Memory Banks	256×16 KB
Frequency	1 GHz

Table 7.3: System parameters for TX2 and PROMISE.

the output of the processed images in comparison to the baseline. We reject configurations with violation rate exceeding 5%.

For our baseline comparison, we map all computations to FP32 with no approximations.

7.3.3 Hardware Platform

Similar to section 6.4.1, for our evaluation we assume a modern SoC architecture with CPUs, GPUs and accelerators that communicate via global shared memory. The specific system we model is the NVIDIA Jetson Tegra TX2 developer kit [116], augmented with a PROMISE accelerator.

We chose a split approach where we collect profiler performance and energy numbers from direct execution on the GPU and simulator results for operations mapped to PROMISE. While a cycle-accurate CPU-GPU-PROMISE simulator would be an alternate approach to

model the SoC, this is infeasible since: 1) open-source GPU simulators (such as GPGPU-Sim) do not support linking against external dynamically linked libraries such as cuBLAS and cuDNN (required for supporting high performance DNN computations), and 2) simulator execution is orders of magnitude slower than real hardware, thus making it infeasible to run heavier computations.

7.3.4 Compilation Flow

ApproxTuner is implemented on top of the HPVM-ApproxHPVM infrastructure. Our DNN benchmarks are written in Keras [135], a popular framework for developing neural network models. The existing ApproxHPVM Keras frontend is invoked to automatically translate from Keras to ApproxHPVM IR. The image processing benchmarks are written in C extended with a set of C functions corresponding to tensor intrinsics. From this tensor-level representation, the existing HPVM C frontend is invoked to compile to ApproxHPVM IR.

We extend the infrastructure with an additional backend, that translates the tensor intrinsics to our runtime. This backend handles translation of only the tensor intrinsics, and translation of all other HPVM constructs (e.g., internal nodes, dataflow edges) is handled by the existing HPVM infrastructure (section 4). Our runtime is implemented on top of NVIDIA's cuDNN library and also includes custom CUDA kernels for certain tensor operations. The runtime includes the implementations of tensor operations along with approximate counterparts introduced in section 7.2.1.

Note that our runtime includes custom handwritten CUDA kernels for convolution since our approximations are based on our custom implementation. While cuDNN convolution is 2-3x faster than our handwritten convolution kernel, the source code is not open-source and hence cannot be used to define approximate versions.

7.3.5 Dynamic Approximation Tuning Experiments

We execute the programs over their respective data set on the target platform. Note that we disable Dynamic Voltage and Frequency Scaling (DVFS) to avoid variations in measurements. Across different runs for each benchmark, we invoke the runtime with a monotonically increasing target speedup value and observe if the measured speedup matches the expected speedup. For each DNN run, we include 5000 images that are split across 10 batches of 500 images. For each image benchmark run, we include 3000 images that are split across 16 batches of 250 images. We do batching because of memory constraints. For

each speedup value, we run the configuration(s) indicated by the runtime controller (refer to section 7.2.5) per batch and average across batches. Across runs, we see negligible standard deviation in runtime measurements.

When targeting configurations that contain software only approximations, the tensor operations are executed directly on the target hardware. When targeting the PROMISE hardware approximation option, we invoke the PROMISE functional simulator for the accuracy of the tensor operations and the PROMISE timing simulator, whose result is added to the application time.

7.3.6 Autotuning Experiments

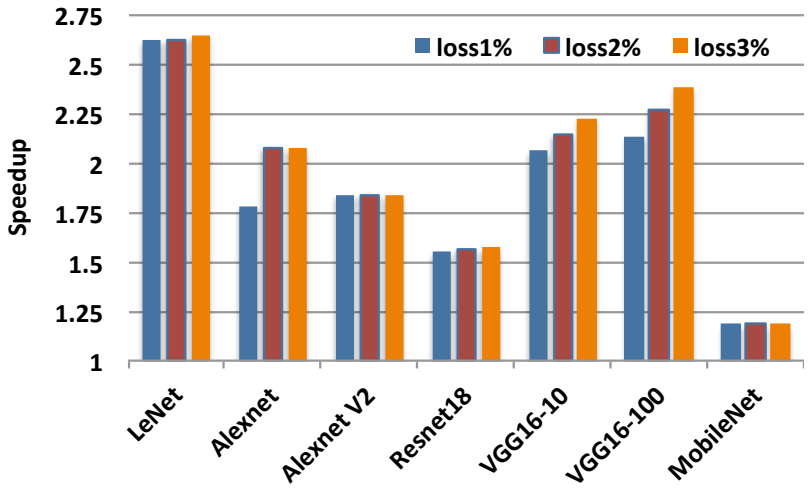
We run our development-time approximation tuning on a server with NVIDIA V100 GPU with 5120 cores and 16GB HBM2 global memory. For federated tuning, we use a combination of experimental and analytical timing since the times for execution on PROMISE accelerator are returned by the PROMISE timing model while the end-to-end accuracy is validated through the provided functional model. For tensor operations scheduled on the GPU, we use the measured time on hardware and for computations mapped to PROMISE, we query the timing model.

7.4 EVALUATION

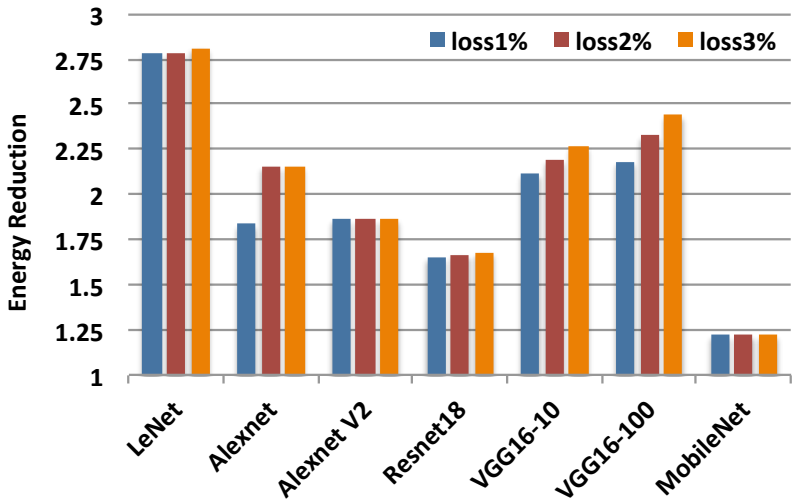
This section presents an evaluation of ApproxTuner. We seek to answer the following questions:

- What are the performance and energy benefits enabled by the development-time autotuning, using hardware-independent approximations?
- Can ApproxTuner do an install-time autotuning to map to additional available hardware-specific approximation choices?
- Can ApproxTuner adapt to runtime resource variations by leveraging dynamic approximation tuning to maintain a target level of application responsiveness?

We refer to development-time generated configurations, that can only include hardware-independent approximations, also as hardware-independent, and install-time generated configurations that may additionally include hardware-specific approximations as hardware-specific.



(a)



(b)

Figure 7.7: (a) Speedups and (b) energy reductions achieved using hardware-independent approximations for $Loss_{1\%}$, $Loss_{2\%}$, $Loss_{3\%}$.

7.4.1 Performance and Energy Evaluation of Development Time Tuning

Figures 7.7a and 7.7b show the performance and energy benefits achieved by the best autotuner-generated hardware-independent configuration, for all DNN benchmarks, for the $Loss_{1\%}$, $Loss_{2\%}$ and $Loss_{3\%}$ experiments. The baseline is the non-approximate computation using 32-bit floating point (FP32) on the GPU.

Each configuration is a set of values for the knobs that control the level of approximation. Using hardware-independent approximations, the autotuner is able to find configurations that achieve speedup ranging from 1.18x to 2.64x and energy reduction from 1.22x to 2.80x,

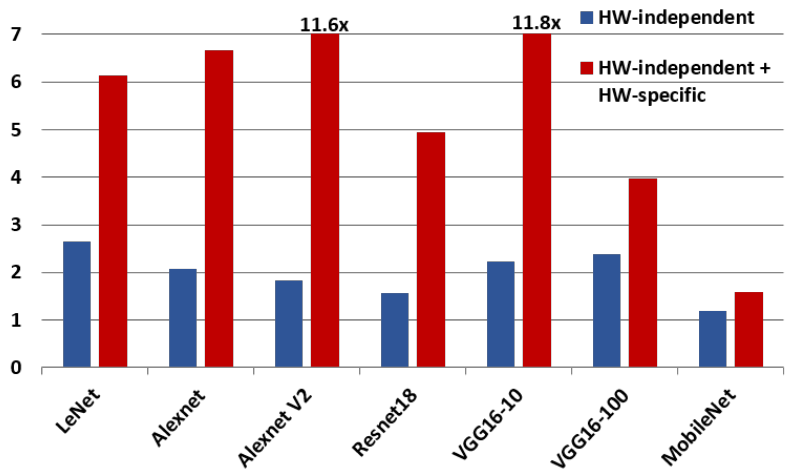


Figure 7.8: Speedups with hardware-independent approximations v.s. speedups with hardware-independent and hardware-specific approximations.

compared with the baseline. The effect of increasing the accuracy loss threshold varies. For example, for VGG-16-100, VGG-16-10 and Alexnet we see increasingly better speedup and energy reduction, while for MobileNet, Resnet, Lenet and Alexnet2 there is no effect on the resulting benefits.

In some DNNs, different approximation choices that result in accuracy loss exceeding the $Loss_{1\%}$ result in an acceptable accuracy loss, within the next $Loss_{2\%}$ threshold, leading to performance and energy benefits. In other DNNs, different approximation choices lead to dramatic increase in accuracy loss, which exceeds any acceptable loss threshold. Some DNNs such as Lenet, are very approximable even with a small accuracy loss threshold. More detailed explanation and insights are provided in section 7.5.

7.4.2 Benefits of Install Time Tuning

Figure 7.8 compares the performance and energy benefits achieved by the best hardware-specific configuration, generated after the retuning step, to the best hardware-independent configuration. We observe a significant improvement in the achieved speedup, from a geometric mean of 1.9x to 5.6x, and energy reduction, from 2.0 to 5.9x, over the hardware-independent configurations for all DNNs except MobileNet, which only improves by 0.34x for both speedup and energy reduction. MobileNet is not very amenable to our approximation techniques, as we are able to approximate very few of its layers, leading to lower performance and benefits for both hardware-independent and hardware-specific configurations. We elaborate more in section 7.5.

These results indicate that install-time autotuning can take advantage of additional hardware-specific approximation techniques to greatly improve upon (already good) speedup and energy reductions obtained from hardware-independent techniques alone. Note that the install-time autotuning achieves this by building on the approximate Pareto-optimal curves created by portable, development-time autotuning.

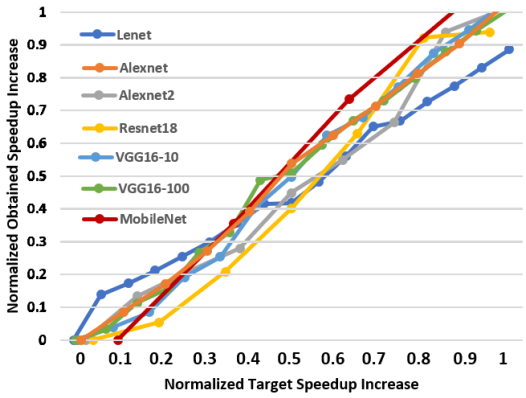
7.4.3 Dynamic Approximation Tuning

For this experiment, where the goal is maintaining a certain level of application responsiveness, we use the control strategy 2 detailed in section 7.2.5. There are multiple conditions that could cause a perceived system slowdown; e.g., system load variation or frequency scaling due to a power cap. The ability to respond to such situations stems from being able to achieve a target speedup at the application level equal to the system slowdown. Thus, for a range of target speedups simulating system slowdowns, we measure the application response. For each DNN in each experiment, below, we limit the target speedup to the maximum observed speedup for the best configuration, since a higher speedup is not achievable by definition. The question, therefore, is how well the runtime system can adapt to changing targets within the achievable range (for each benchmark).

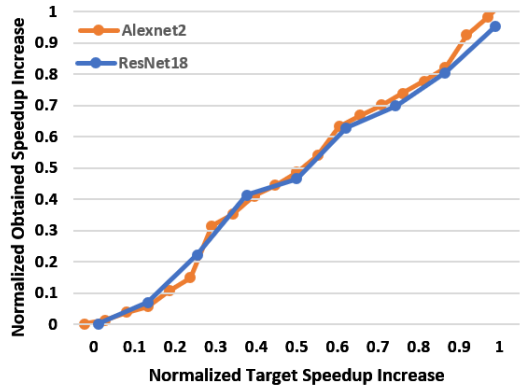
Figures 7.9a, 7.9b and 7.9c show the results of obtained speedup for our benchmarks vs the target speedup, and Figures 7.10a and 7.10b show the corresponding quality loss. For all the speedup graphs, the x-axis is the (normalized) target speedup increase over the baseline configuration, and y-axis is the (normalized) obtained speedup increase over the baseline configuration. Both axes are normalized to the maximum achievable speedup increase for each benchmark, e.g., determined using data from figure 7.7a for DNNs. If s is the target speedup (x-axis) or the obtained speedup (y-axis), then we compute the normalized speedup *increase* over the baseline speedup as $\frac{s-1}{s_{max}-1}$, where we subtract 1 because the baseline configuration always has a speedup of 1.

Figures 7.9a and 7.10a show the normalized obtained speedup increase and accuracy loss for all DNN benchmarks, using only hardware-independent approximations. In figure 7.9a, we observe that the DNNs are mostly able to adapt to the target speedup, since they are close to the diagonal.

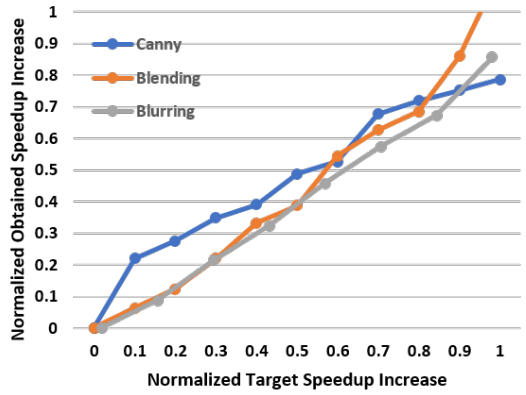
Figure 7.10a shows that accuracy loss tends to increase as the target speedup increases, with the exceptions of VGG-16-10 and VGG-16-100. For both these cases, the behavior of two configurations was different on the calibration set and the test set. For VGG-16-10, two configurations C_1 and C_2 in the Pareto optimal set are ordered such that C_2 has higher speedup but worse accuracy loss according to the calibration set measurements, but



(a)

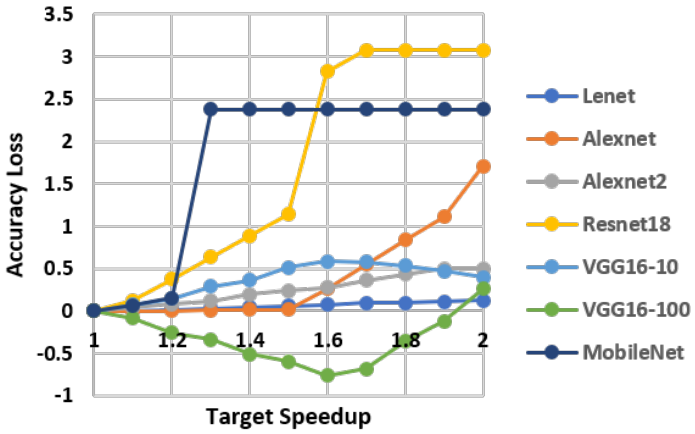


(b)

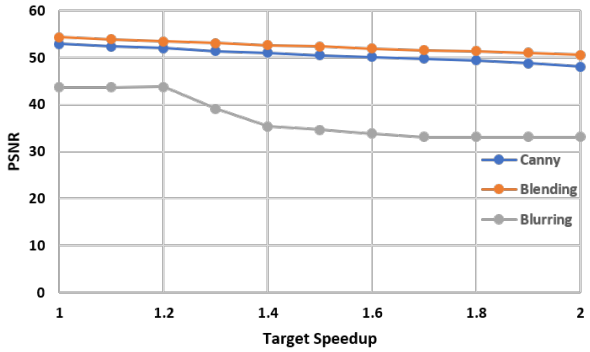


(c)

Figure 7.9: Normalized obtained speedup increase over respective benchmark baseline for target speedup increase. Obtained and target speedup increase are normalized to each benchmark's maximum achievable speedup − 1. (a) DNNs with hardware-independent configurations. (b) DNNs with hardware-specific configurations. (c) Image benchmarks with hardware-independent configurations.



(a) DNN accuracy loss.



(b) PSNR of image benchmarks.

Figure 7.10: Quality loss over respective benchmark baseline with hardware-independent approximations, for target speedup.

C_2 outperforms C_1 on the test set. As C_2 is gradually selected with higher probability, the accuracy loss gradually decreases. A similar situation occurs in VGG-16-100, where a configuration shows better end-to-end accuracy on the unseen test data compared to calibration data. This behavior is uncommon but can manifest in certain scenarios. A final observation is the steep increase of accuracy loss for MobileNet. It occurs because in the search for configurations during the autotuning, the autotuner found configurations with very little performance benefit for much higher accuracy loss, which still constitute Pareto-optimal points. Aiming for the highest possible speedup results in getting the configuration with the worst accuracy loss, and due to the small performance difference between these configurations we observe the steep increase.

Figure 7.9b shows the speedup increase for Alexnet2 and Resnet using hardware-specific approximations, for a target speedup increase (again normalized to the maximum achievable

Benchmark	GPU
Lenet-5	samp-50%:1 perf-50%:1 FP16:2
Alexnet	FP16:2 samp-50%:2 samp-25%:2
Alexnet2	FP16:4 perf-50%:1 samp-50%:1 perf-33%:1
Resnet-18	FP16:11 perf-50%:5 perf-33%:4 samp-25%:2
VGG-16-10	FP16:5 perf-50%:4 perf-33%:2 samp-50%:4
VGG-16-100	FP16:3 perf-50%:2 samp-50%:8 perf-33%:2
MobileNet	FP16:21 perf-50%:5 perf-33%:2

(a) Approximation knobs for GPU configuration with maximum speedup and accuracy loss < 3.

Benchmark	GPU + Promise
Lenet-5	samp-50%:1 P5:1 FP16:2
Alexnet	FP16:2 P3:2 P6:1 P7:1
Alexnet2	FP16:2 P6:3 P5:1 P3:1
Resnet-18	FP16:6 P7:6 perf-50%:2 P6:4 P5:3 P3:1
VGG-16-10	FP16:4 P5:3 P6:2 P7:2 P3:3 samp-50%:1
VGG-16-100	FP16:4 P5:4 samp-50%:3 P3:2 perf-33%:1 perf-50%:1
MobileNet	FP16:17 P5:2 P3:3 P6:3 P4:2 perf-33%:1

(b) Approximation knobs for GPU and promise configuration with maximum speedup and accuracy loss < 3.

Table 7.4: Selected Approximation Knobs for Evaluated DNNs.

additional speedup, obtained from Figure 7.8). Alexnet2 and Resnet are representative of those with the highest and those with lower to moderate benefits from retuning with hardware-specific approximations. We observe that the DNNs are able to adapt to the target speedup, and doing so on the much wider range of target speedups enabled by the hardware-specific configurations.

Similarly, figures 7.9c and 7.10b show the normalized obtained speedup increase and PSNR for all image benchmarks, using hardware-independent approximations. Similar to the DNNs, figure 7.9c shows the ability to adapt to the target speedup requirement. Figure 7.10b shows that PSNR tends to decrease as target speedup increases.

7.5 SENSITIVITY TO APPROXIMATION TECHNIQUES

In this section, we discuss the impact of the various approximations on the evaluated benchmarks. Table 7.4a includes the knob settings corresponding to the best configuration for each DNN benchmark considering only hardware-independent approximations. Table 7.4b includes the knob settings of the best configuration after the install-time retuning that maps certain convolution and fully-connected layers to PROMISE.

General trends. We find that all DNNs are very amenable to FP16 compute with minimal accuracy loss and providing benefits ranging from 1-1.6x. The filter sampling and output

perforation also use FP16 precision (since FP16 has minimal accuracy impact) and hence the benefits these techniques provide are in addition to those provided by merely FP16 compute. The individual benefits achieved by mapping a layer to filter sampling range from 1-2.6x and for perforation range from 1-2.6x. We summarize some key insights per DNNs:

Alexnet: With Alexnet, we observe a few interesting insights: i) the model is very amenable to the filter sampling approximation but very susceptible to perforated convolutions. None of the layers in Alexnet (for all configurations) map to the perforated convolutions approximation while all layers (in various configurations) are amenable to filter sampling. ii) Layer 2 and Layer 4 are relatively more susceptible to approximation since only 25% sampling could be mapped to these layers (without losing significant accuracy). Table 7.4b shows that after retuning a lot of layers can be mapped to PROMISE which shows that Alexnet is also very amenable to the hardware approximation offered by PROMISE.

MobileNet: For the best configuration, only 7 layers (out of 28) are mapped to approximation techniques. Since 75% layers are not found to be approximable by the approximation tuner (except for FP16), MobileNet gives the least benefits (1.18x speedup) across all the benchmarks. FP16 also doesn't provide significant benefits for MobileNet.

We find another observation with MobileNet: across all 85 configurations, layer 5, 9, 10 can only be mapped to column perforation but not row perforation.

Resnet: Across all configurations, 7 of the 21 convolution layers are not mapped to any approximation. Interestingly, 4 of the 21 layers are only mapped to 33% perforation and all 4 perforations start at different start offsets. Such observations confirm the hypothesis that varying start offsets with perforation (and sampling) combine well together.

Lenet: We find Lenet to be highly approximable. The first layer in Lenet can map to all 10 knobs of perforation and all 6 knobs for filter sampling. Note that after retuning, 2 layers (except for first layer and last layer) can be mapped to PROMISE.

VGG-16-100: Similar to Resnet, we find that 3 layers in VGG-16-100 can only be mapped to column-based perforation while row-based perforation leads to high accuracy loss.

Key Takeaways. For 4 of the 7 DNN benchmarks, the first layer is found susceptible and not approximated. In Resnet and MobileNet, the first 3 layers are never mapped to any approximation.

These insights show that different benchmarks and operations within benchmarks have highly varying susceptibility to the different approximations. Since ApproxTuner uses development-time and install-time approximation tuning, it is able to discover good configurations for most benchmarks.

CHAPTER 8: CONCLUSION AND FUTURE DIRECTIONS

8.1 CONCLUSION

The slow down of Moore's law and the end of Denard's scaling has given rise to heterogeneous systems, with custom hardware components with superior performance and energy efficiency. However, the diversity of the underlying hardware components also gives rise to new programmability challenges. We identify the root causes of the programmability challenges as

1. diverse parallelism models
2. diverse memory systems
3. diverse instruction sets
4. diverse approximation methods, in software and hardware

and we propose compiler solutions for addressing the object code and performance portability challenge of heterogeneous systems, as well as techniques for exploiting their superior performance and energy efficiency. Specifically, we make the following contributions:

1. We proposed HPVM, a portable parallel program representation for a wide range of parallel hardware. HPVM is a based on a hierarchical dataflow graph with side effects. Based on the HPVM representation, we define an HPVM prototype on top of LLVM that successfully serves as a virtual ISA, a compiler IR and a Runtime Representation. We implement translators for NVIDIA's GPUs, Intel's AVX vector units, and multicore X86-64 processors. We implement node fusion as a compiler transformation and show that tiling is also captured in the HPVM representation. Our evaluation shows that HPVM achieves functional and performance portability across these classes of hardware, that the optimizations can achieve significant performance gains, and the HPVM representation is able to support flexible scheduling policies both statically and dynamically.
2. We extended HPVM to ApproxHPVM, introducing support for domain specific information and approximation in the HPVM framework. ApproxHPVM IR extends the HPVM IR in two key ways: (1) with tensor operations, thereby adding domain specific information at the IR level, and (2) accuracy metrics, that capture the allowable difference between the exact and approximate execution of the tensor operations.

3. We proposed ApproxTuner, an extension of ApproxHPVM that addresses the challenges that arise on heterogeneous systems at the edge. ApproxTuner utilizes a three step approximation tuning strategy: development time, install time, and dynamic tuning, thus ensuring object code portability, practical tuning times, and fast dynamic approximation tuning. Our evaluation shows that ApproxTuner achieves good performance and energy benefits during development time tuning, which are further improved after the install time tuning phase, and that the system can adapt as needed to dynamic variations through dynamic approximation tuning.

8.2 FUTURE DIRECTIONS

8.2.1 HPVM Extensions for Complex Memory Systems

The HPVM abstractions successfully abstract away he diversities in underlying hardware ISAs and parallelism models exposed by a variety of different hardware targets. However, although the hierarchical dataflow graph abstraction is able to capture tiling and private memories at any level of a memory system, there is a wide margin for improvement in the memory abstractions in HPVM. Systems like Sequoia [72] and Legion [70] have rich memory abstractions, for example Legion allows tasks to specify logical regions of data that will be accessed, specifying the privileges and coherence of that access. In HPVM, much more limited properties of accessed memory are specified, offering fewer opportunities for data partitioning, movement and optimization to the HPVM runtime. Identifying a set of information pertinent to memory system optimizations and introducing them to the HPVM model will enable optimizations for systems with complex memory hierarchies, where the cost of communication is significant, and may even overcome the cost of computation.

8.2.2 Introducing Cycles in the HPVM Dataflow Graph

The HPVM representation is designed as a hierarchical dataflow graph, that is required to be *acyclic*. The acyclic property is beneficial albeit limiting in certain ways, e.g., we study error propagation through the acyclic graph, however it would be increasingly difficult to study how the error evolves in a dataflow graph with cycles. However, there are important applications that would benefit from the presence of cycles in the dataflow graph. At the moment we rely on the host code to repeatedly launch the dataflow graph in a loop to represent such applications. Introducing a "backwards dataflow edge" to the dataflow graph, effectively creating a cycle, would represent a change in the HPVM model that poses research

challenges, but enables a wide range of applications to be represented as dataflow graphs in HPVM.

8.2.3 Compiler Optimizations on The HPVM IR

The HPVM representation has been used as a compiler IR in performing the *node fusion* optimization. However, currently the optimization is performed without estimating its profitability, and can be enhanced with a model or tuning driven component to select the fusion targets. Additionally, a wider set of transformations can be designed on top of the dataflow graph representation, both target independent and target dependent. For example, the opposite transformation, node splitting can be implemented, to be performed as deemed beneficial by an appropriate (target specific) cost model, but as an target independent graph transformation. More generally, a wide set of transformations on top of the dataflow graph will enable easier and better code generation for a range of target hardware. Different hardware targets may have different characteristics and target specific translators for them may need to transform the dataflow graph. Identifying and designing these transformations on the hardware independent dataflow graph representation is an open problem, that will enable all translators to benefit from them and improve the performance of generated code.

8.2.4 Efficient Scheduling Policy in HPVM

In the evaluation section of the HPVM prototype (section 5.3) we demonstrate that HPVM enables flexible runtime scheduling decisions. The HPVM representation allows the runtime scheduler to dynamically select between different native versions of generated code for HPVM dataflow nodes to compensate for load variations at runtime. However, implementation of more sophisticated scheduling policies within the framework is an open problem, especially in combination with performing compiler transformations on the HPVM representation and memory model extensions.

8.2.5 Extension of HPVM IR to Other Domains/Classes of Algorithms

We have currently extended the HPVM IR with domain specific information in the form of high level operations. Direct representation of high level operations at the IR level is a design choice that enables mapping to efficient hardware and utilization of domain specific optimizations and approximations. The current set of operations captures tensor operations for the domains of convolutional neural networks and image processing. However, extending

to other domains, or classes of algorithms within a domain is not straightforward. Identifying a minimal but sufficient set of operations and any other information required to represent computations within a domain and developing domain specific approximation techniques are interesting and open problems in this direction.

REFERENCES

[1] M. D. Hill and M. R. Marty, "Amdahl's Law in the Multicore Era," Computer, vol. 41, no. 7, p. 33–38, July 2008. [Online]. Available: https://doi.org/10.1109/MC.2008.209

[2] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark Silicon and the End of Multicore Scaling," in Proceedings of the 38th Annual International Symposium on Computer Architecture, ser. ISCA '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/2000064.2000108 p. 365–376.

[3] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz, "Understanding Sources of Inefficiency in General-Purpose Chips," SIGARCH Comput. Archit. News, vol. 38, no. 3, p. 37–47, June 2010. [Online]. Available: https://doi.org/10.1145/1816038.1815968

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," in Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on. IEEE, 2017, pp. 1–12.

[5] P. Greenhalgh, "Big.LITTLE Processing with ARM Cortex-A15 & Cortex-A7," 2011.

[6] Qualcomm, "Snapdragon 865 5G Mobile Platform," https://www.qualcomm.com/products/snapdragon-865-5g-mobile-platform, 2019.

[7] Qualcomm, "Snapdragon 865 5G Mobile Platform Specifications & Features," https://www.qualcomm.com/media/documents/files/qualcomm-snapdragon-865-5g-mobile-platform-product-brief.pdf, 2019.

[8] Qualcomm, "Snapdragon 865 5G Mobile Platform," https://developer.qualcomm.com/software/hexagon-dsp-sdk/dsp-processor, 2019.

[9] National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign, "Blue Waters," https://bluewaters.ncsa.illinois.edu/.

[10] Lawrence Livermore National Laboratory, "Sierra," https://computing.llnl.gov/computers/sierra.

[11] Texas Advanced Computing Center, "Frontera," https://www.tacc.utexas.edu/systems/frontera.

[12] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," 02 2018, pp. 620–629.

[13] H. Li, K. Ota, and M. Dong, "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing," IEEE Network, vol. 32, no. 1, pp. 96–101, Jan 2018.

[14] D. Azariadi, V. Tsoutsouras, S. Xydis, and D. Soudris, "ECG signal analysis and arrhythmia detection on IoT wearable medical devices," in 2016 5th International Conference on Modern Circuits and Systems Technologies (MOCAST), May 2016, pp. 1–4.

[15] M. Mehrabani, S. Bangalore, and B. Stern, "Personalized speech recognition for Internet of Things," in 2015 IEEE 2nd World Forum on Internet of Things (WF-IoT), Dec 2015, pp. 369–374.

[16] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," IEEE Internet of Things Journal, vol. 3, no. 5, pp. 637–646, 2016.

[17] International Data Corporation, "The Growth in Connected IoT Devices Is Expected to Generate 79.4ZB of Data in 2025," https://www.idc.com/getdoc.jsp?containerId=prUS45213219, 2019.

[18] P. Stanley-Marbell, A. Alaghi, M. Carbin, E. Darulova, L. Dolecek, A. Gerstlauer, G. Gillani, D. Jevdjic, T. Moreau, M. Cacciotti et al., "Exploiting Errors for Efficiency: A Survey from Circuits to Algorithms," arXiv preprint arXiv:1809.05859, 2018.

[19] P. Srivastava, M. Kang, S. K. Gonugondla, S. Lim, J. Choi, V. Adve, N. S. Kim, and N. Shanbhag, "PROMISE: An End-to-End Design of a Programmable Mixed-Signal Accelerator for Machine-Learning Algorithms," in 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). IEEE, 2018.

[20] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[21] NVIDIA, "NVIDIA TESLA V100 GPU Architecture," https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf.

[22] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A Language for Streaming Applications," ser. International Conference on Compiler Construction, 2002.

[23] J. Auerbach, D. F. Bacon, P. Cheng, and R. Rabbah, "Lime: a Java-Compatible and Synthesizable Language for Heterogeneous Architectures," in Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, ser. OOPSLA '10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869469 pp. 89–108.

[24] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "PetaBricks: A Language and Compiler for Algorithmic Choice," ser. PLDI, 2009.

[25] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia et al., "Machine Learning at Facebook: Understanding Inference at the Edge," in 2019 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE, 2019, pp. 331–344.

[26] M. Okamoto, K. Yamashita, H. Kasahara, and S. Narita, "Hierarchical macro-dataflow computation scheme," in IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing. Proceedings. IEEE, 1995. [Online]. Available: https://doi.org/10.1109/pacrim.1995.519406

[27] Y. Wada, A. Hayashi, T. Masuura, J. Shirako, H. Nakano, H. Shikano, K. Kimura, and H. Kasahara, A Parallelizing Compiler Cooperative Heterogeneous Multicore Processor Architecture. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011.

[28] N. Benoit and S. Louise, "Using an Intermediate Representation to Map Workloads on Heterogeneous Parallel Systems," in 24th Euromicro Conference, 2016.

[29] D. Khaldi, P. Jouvelot, F. Irigoin, and C. Ancourt, "SPIRE: A Methodology for Sequential to Parallel Intermediate Representation Extension," ser. CPC, 2012.

[30] T. B. Schardl, W. S. Moses, and C. E. Leiserson, "Tapir: Embedding Fork-Join Parallelism into LLVM's Intermediate Representation," ser. PPoPP, 2017.

[31] A. Sampson, A. Baixo, B. Ransford, T. Moreau, J. Yip, L. Ceze, and M. Oskin, "ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing," in U. Washington, Tech. Rep. UW-CSE- 15-01-01, 2015.

[32] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate Data Types for Safe and General Low-Power Computation," ser. PLDI, 2011.

[33] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware," in Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2509136.2509546 p. 33–52.

[34] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: Reliability- and Accuracy-Aware Optimization of Approximate Computational Kernels," in ACM SIGPLAN Notices, vol. 49, no. 10. ACM, 2014, pp. 309–328.

[35] H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard, "Dynamic Knobs for Responsive Power-Aware Computing," ser. ASPLOS, 2011.

[36] H. Hoffmann, "JouleGuard: Energy Guarantees for Approximate Applications," in Proceedings of the 25th Symposium on Operating Systems Principles. ACM, 2015, pp. 198–214.

[37] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-Tuning Approximation for Graphics Engines," in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-46. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2540708.2540711 pp. 13–24.

[38] W. Baek and T. M. Chilimbi, "Green: A Framework for Supporting Energy-Conscious Programming Using Controlled Approximation," ser. PLDI, 2010.

[39] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using Code Perforation to Improve Performance, Reduce Energy Consumption, and Respond to Failures," Massachusetts Institute of Technology, Tech. Rep. MIT-CSAIL-TR-2009-042, 2009.

[40] S. Li, S. Park, and S. Mahlke, "Sculptor: Flexible Approximation with Selective Dynamic Loop Perforation," in Proceedings of the 2018 International Conference on Supercomputing, ser. ICS '18. New York, NY, USA: ACM, 2018. [Online]. Available: https://doi.org/10.1145/3205289.3205317 p. 341–351.

[41] LLVM Project, "LLVM Language Reference Manual," 2003. [Online]. Available: http://llvm.org/docs/LangRef.html

[42] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe, "Language and Compiler Support for Auto-Tuning Variable-Accuracy Algorithms," in Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on. IEEE, 2011, pp. 85–96.

[43] A. Gulli and S. Pal, Deep Learning with Keras. Packt Publishing Ltd, 2017.

[44] R. S. Nikhil, "The parallel programming language Id and its compilation for parallel machines," ser. IJHSC, 1993.

[45] D. Culler, S. Goldstein, K. Schauser, and T. Voneicken, "TAM - A Compiler Controlled Threaded Abstract Machine," 1993.

[46] E. A. Ashcroft and W. W. Wadge, "LUCID, a Nonprocedural Language with Iteration," Commun. ACM, 1977.

[47] Google, "Google Cloud Dataflow," 2013. [Online]. Available: https://cloud.google.com/dataflow/

[48] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Supporting Stateful Tasks in a Dataflow Graph," in Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques, ser. PACT '12. New York, NY, USA: ACM, 2012, pp. 435–436.

[49] V. Gajinov, S. Stipic, O. S. Unsal, T. Harris, E. Ayguadé, and A. Cristal, "Integrating Dataflow Abstractions into the Shared Memory Model," in 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing, Oct 2012, pp. 243–251.

[50] J. Zhou and B. Demsky, "Bamboo: A Data-Centric, Object-Oriented Approach to Many-core Software," in Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 388–399.

[51] H. Foundation, "HSAIL," 2015. [Online]. Available: http://www.hsafoundation.com/standards/

[52] Khronos Group, "SPIR-V 1.5 Specification," https://www.khronos.org/registry/spir-v/specs/unified1/SPIRV.pdf, 2019.

[53] T. Miyamoto, S. Asaka, H. Mikami, M. Mase, Y. Wada, H. Nakano, K. Kimura, and H. Kasahara, "Parallelization with Automatic Parallelizing Compiler Generating Consumer Electronics Multicore API," in 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications. IEEE, dec 2008.

[54] N. Benoit and S. Louise, "Extending GCC with a Multi-grain Parallelism Adaptation Framework for MPSoCs," in 2nd Int'l Workshop on GCC Research Opportunities, 2010.

[55] A. J. Sabne, P. Sakdhnagool, S. Lee, and J. S. Vetter, "Understanding Portability of a High-Level Programming Model on Contemporary Heterogeneous Architectures," IEEE Micro, vol. 35, no. 4, 7 2015.

[56] S. Lee and J. S. Vetter, "OpenARC: Open Accelerator Research Compiler for Directive-Based, Efficient Heterogeneous Computing," in Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing, ser. HPDC '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2600212.2600704 p. 115–120.

[57] AMD, "AOMP," https://github.com/ROCm-Developer-Tools/aomp.

[58] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, "Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement," in Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing, ser. LCPC'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 172–187.

[59] D. Majeti and V. Sarkar, "Heterogeneous Habanero-C (H2C): A Portable Programming Model for Heterogeneous Processors," ser. IPDPS Workshop, 2015.

[60] L. Chang, A. Dakkak, C. I. Rodrigues, and W. mei Hwu, "Tangram: a High-level Language for Performance Portable Code Synthesis," ser. MULTIPROG 2015, 2015.

[61] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A Compiler Architecture for Performance-Oriented Embedded Domain-Specific Languages," ser. ACM TECS, 2014.

[62] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "MLIR: A Compiler Infrastructure for the End of Moore's Law," 2020.

[63] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam, PVM: A Users' Guide and Tutorial for Networked Parallel Computing. MIT press, 1994.

[64] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," Concurrency and Computation: Practice and Experience, 2011.

[65] Qualcomm Technologies, Inc., "MARE: Enabling Applications for Heterogeneous Mobile Devices," White Paper, Tech. Rep., 2014.

[66] T. Mattson, R. Cledat, Z. Budimlic, V. Cave, S. Chatterjee, B. Seshasayee, W. R. van der, and V. Sarkar, "OCR: The Open Community Runtime Interface," Tech. Rep., 2015.

[67] T. Ben-Nun, M. Sutton, S. Pai, and K. Pingali, "Groute: An Asynchronous Multi-GPU Programming Model for Irregular Computations," in Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '17. New York, NY, USA: ACM, 2017, pp. 235–248.

[68] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming," in Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT), ser. PACT '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 138–149.

[69] Z. Budimlic, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Tasirlar, "Concurrent Collections," Scientific Programming, vol. 18, no. 3-4, pp. 203–217, 2010.

[70] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing Locality and Independence with Logical Regions," ser. SC, 2012.

[71] S. Treichler, M. Bauer, and A. Aiken, "Realm: An Event-Based Low-Level Runtime for Distributed Memory Architectures," in Proceedings of the 23rd International Conference on Parallel Architectures and Compilation, ser. PACT '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2628071.2628084 p. 263–276.

[72] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan, "Sequoia: Programming the Memory Hierarchy," in Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, ser. SC '06. New York, NY, USA: Association for Computing Machinery, 2006. [Online]. Available: https://doi.org/10.1145/1188455.1188543 p. 83–es.

[73] M. Bauer, J. Clark, E. Schkufza, and A. Aiken, "Programming the Memory Hierarchy Revisited: Supporting Irregular Parallelism in Sequoia," in Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '11. New York, NY, USA: Association for Computing Machinery, 2011. [Online]. Available: https://doi.org/10.1145/1941553.1941558 p. 13–24.

[74] B. Boston, A. Sampson, D. Grossman, and L. Ceze, "Probability Type Inference for Flexible Approximate Programming," in OOPSLA. ACM, 2015, pp. 470–487.

[75] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze et al., "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning," in 13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18), 2018, pp. 578–594.

[76] N. Rotem, J. Fix, S. Abdulrasool, S. Deng, R. Dzhabarov, J. Hegeman, R. Levenstein, B. Maher, N. Satish, J. Olesen, J. Park, A. Rakhov, and M. Smelyanskiy, "Glow: Graph Lowering Compiler Techniques for Neural Networks," CoRR, vol. abs/1805.00907, 2018. [Online]. Available: http://arxiv.org/abs/1805.00907

[77] "Domain-specific compiler for linear algebra to optimize tensorflow computations," https://www.tensorflow.org/xla/overview, 2018.

[78] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "OpenTuner: An Extensible Framework for Program Autotuning," in Proceedings of the 23rd international conference on Parallel architectures and compilation. ACM, 2014, pp. 303–316.

[79] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning Algorithmic Choice for Input Sensitivity," in Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, ser. PLDI '15. New York, NY, USA: ACM, 2015. [Online]. Available: http://doi.acm.org/10.1145/2737924.2737969 pp. 379–390.

[80] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, and S. Amarasinghe, "SiblingRivalry: Online Autotuning Through Local Competitions," in Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, ser. CASES '12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2380403.2380425 pp. 91–100.

[81] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. Rinard, "Quality of Service Profiling," ser. ICSE, 2010.

[82] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing Performance vs. Accuracy Trade-offs With Loop Perforation," in Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. ACM, 2011, pp. 124–134.

[83] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in ACM SIGARCH Computer Architecture News, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[84] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An Instruction Set Architecture for Neural Networks," in ACM SIGARCH Computer Architecture News, vol. 44, no. 3. IEEE Press, 2016, pp. 393–405.

[85] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, "ShiDianNao: Shifting Vision Processing Closer to the Sensor," in ACM SIGARCH Computer Architecture News, vol. 43, no. 3. ACM, 2015, pp. 92–104.

[86] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun et al., "DaDianNao: A Machine-Learning Supercomputer," in Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. IEEE Computer Society, 2014, pp. 609–622.

[87] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural Acceleration for General-Purpose Approximate Programs," in Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, ser. MICRO-45. USA: IEEE Computer Society, 2012. [Online]. Available: https://doi.org/10.1109/MICRO.2012.48 p. 449–460.

[88] R. St Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-Purpose Code Acceleration with Limited-Precision Analog Computation," ACM SIGARCH Computer Architecture News, vol. 42, no. 3, pp. 505–516, 2014.

[89] D. Lin, S. Talathi, and S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in International Conference on Machine Learning, 2016, pp. 2849–2858.

[90] C. Sakr, Y. Kim, and N. Shanbhag, "Analytical Guarantees on Numerical Precision of Deep Neural Networks," in International Conference on Machine Learning, 2017, pp. 3007–3016.

[91] C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough, "Precimonious: Tuning Assistant for Floating-Point Precision," in High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for. IEEE, 2013, pp. 1–12.

[92] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic Optimization of Floating-Point Programs with Tunable Precision," ser. PLDI, 2014.

[93] M. Rinard, "Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks," ser. ICS, 2006.

[94] S. Chakradhar, A. Raghunathan, and J. Meng, "Best-Effort Parallel Execution Framework for Recognition and Mining Applications," ser. IPDPS, 2009.

[95] J. Meng, A. Raghunathan, S. Chakradhar, and S. Byna, "Exploiting the Forgiving Nature of Applications for Scalable Parallel Execution," ser. IPDPS, 2010.

[96] S. Misailovic, D. Roy, and M. Rinard, "Probabilistically Accurate Program Transformations," ser. SAS, 2011.

[97] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard, "Randomized Accuracy-Aware Program Transformations for Efficient Approximate Computations," in Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ser. POPL '12. New York, NY, USA: Association for Computing Machinery, 2012. [Online]. Available: https://doi.org/10.1145/2103656.2103710 p. 441–454.

[98] M. Samadi, D. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: Pattern-Based Approximation for Data Parallel Applications," ser. ASPLOS, 2014.

[99] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen, "ApproxHadoop: Bringing Approximations to MapReduce Frameworks," in ASPLOS. ACM, 2015, pp. 383–397.

[100] S. Misailovic, D. Kim, and M. Rinard, "Parallelizing Sequential Programs with Statistical Accuracy Tests," ACM Transactions on Embedded Computing Systems (TECS), vol. 12, no. 2s, p. 88, 2013.

[101] S. Misailovic, S. Sidiroglou, and M. C. Rinard, "Dancing with Uncertainty," in Proceedings of the 2012 ACM workshop on Relaxing synchronization for multicore and manycore scalability. ACM, 2012, pp. 51–60.

[102] S. Campanoni, G. Holloway, G.-Y. Wei, and D. Brooks, "HELIX-UP: Relaxing Program Semantics to Unleash Parallelization," in Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization. IEEE Computer Society, 2015, pp. 235–245.

[103] M. Figurnov, A. Ibraimova, D. Vetrov, and P. Kohli, "Perforated CNNs: Acceleration through Elimination of Redundant Convolutions," in Proceedings of the 30th International Conference on Neural Information Processing Systems, ser. NIPS'16. USA: Curran Associates Inc., 2016. [Online]. Available: http://dl.acm.org/citation.cfm?id=3157096.3157203 pp. 955–963.

[104] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, "MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints," in Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services. ACM, 2016, pp. 123–136.

[105] R. Xu, J. Koo, R. Kumar, P. Bai, S. Mitra, G. Maghanath, and S. Bagchi, "ApproxNet: Content and Contention Aware Video Analytics System for the Edge," arXiv preprint arXiv:1909.02068, 2019.

[106] M. Kotsifakou, P. Srivastava, M. D. Sinclair, R. Komuravelli, V. Adve, and S. Adve, "HPVM: Heterogeneous Parallel Virtual Machine," in Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '18. New York, NY, USA: ACM, 2018. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178493 pp. 68–80.

[107] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," in Proc. Conf. on Code Generation and Optimization, San Jose, CA, USA, Mar 2004, pp. 75–88.

[108] NVIDIA, "PTX: Parallel Thread Execution ISA," 2009, http://docs.nvidia.com/cuda/parallel-thread-execution/index.html.

[109] NVIDIA, "NVVM IR," http://docs.nvidia.com/cuda/nvvm-ir-spec, 2013.

[110] "LIBCLC," https://libclc.llvm.org/.

[111] Khronos Group, "SPIR 1.2 Specification," https://www.khronos.org/registry/spir/specs/spir_spec-1.2.pdf, 2012.

[112] R. Allen and K. Kennedy, Optimizing Compilers for Modern Architectures. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 2002.

[113] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-M. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," Tech. Rep., 2012.

[114] Li-wen Chang, "Personal Communication," August 2015.

[115] H. Sharif, P. Srivastava, M. Huzaifa, M. Kotsifakou, K. Joshi, Y. Sarita, N. Zhao, V. S. Adve, S. Misailovic, and S. Adve, "ApproxHPVM: A Portable Compiler IR for Accuracy-Aware Optimizations," Proc. ACM Program. Lang., vol. 3, no. OOPSLA, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3360612

[116] NVIDIA, "NVIDIA Jetson TX2 Developer Kit," https://www.nvidia.com/en-us/autonomous-machines/embedded-systems-dev-kits-modules, 2018.

[117] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten Digit Recognition with a Back-Propagation Network," in Advances in neural information processing systems, 1990, pp. 396–404.

[118] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification with Deep Convolutional Neural Networks," in Advances in neural information processing systems, 2012, pp. 1097–1105.

[119] W. Yang, "Classification on CIFAR-10/100 and ImageNet with PyTorch," https://github.com/bearpaw/pytorch-classification/blob/master/models/cifar/alexnet.py, 2019.

[120] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[121] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," arXiv 1409.1556, 09 2014.

[122] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," CoRR, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[123] Y. LeCun, C. Cortes, and C. J. Burges, "The MNIST Database of handwritten digits," http://yann.lecun.com/exdb/mnist, 1998.

[124] A. Krizhevsky, "Learning Multiple Layers of Features from Tiny Images," University of Toronto, 05 2012.

[125] P. Wang and J. Cheng, "Accelerating Convolutional Neural Networks for Mobile Applications," in Proceedings of the 24th ACM international conference on Multimedia. ACM, 2016, pp. 541–545.

[126] D. Maier, B. Cosenza, and B. Juurlink, "Local Memory-Aware Kernel Perforation," in Proceedings of the 2018 International Symposium on Code Generation and Optimization, ser. CGO 2018. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3168814 p. 278–287.

[127] H. Wu, "Low Precision Inference on GPU," 2019, gTC. [Online]. Available: https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9659-inference-at-reduced-precision-on-gpus.pdf

[128] Mark Harris, NVIDIA, "Mixed-Precision Programming with CUDA 8," https://devblogs.nvidia.com/mixed-precision-programming-cuda-8/, 2016.

[129] P. Konsor, "Performance Benefits of Half Precision Floats," https://software.intel.com/en-us/articles/performance-benefits-of-half-precision-floats, 2011, accessed: 2019-11-21.

[130] ARM, "Half-precision floating-point number format," https://developer.arm.com/docs/dui0774/e/other-compiler-specific-features/half-precision-floating-point-number-format, 2019, accessed: 2019-11-21.

[131] B. McMahan and D. Ramage, "Federated Learning: Collaborative Machine Learning without Centralized Training Data," https://ai.googleblog.com/2017/04/federated-learning-collaborative.html, April 2017.

[132] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. M. Kiddon, J. Konečný, S. Mazzocchi, B. McMahan, T. V. Overveldt, D. Petrou, D. Ramage, and J. Roselander, "Towards Federated Learning at Scale: System Design," in SysML 2019, 2019, to appear. [Online]. Available: https://arxiv.org/abs/1902.01046

[133] W. Yuan, S. Adve, D. Jones, and R. Kravets, "Design and Evaluation of a Cross-Layer Adaptation Framework for Mobile Multimedia Systems," 01 2003.

[134] Intel, "Intel Image Classification," https://www.kaggle.com/puneet6060/intel-image-classification, 2019.

[135] F. Chollet et al., "Keras," https://keras.io, 2015.